LNCS 10747

ARCoSS

Isil Dillig
Jens Palsberg (Eds.)

Verification, Model Checking, and Abstract Interpretation

19th International Conference, VMCAI 2018
Los Angeles, CA, USA, January 7–9, 2018
Proceedings

Lecture Notes in Computer Science 10747

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison, UK
Takeo Kanade, USA
Josef Kittler, UK
Jon M. Kleinberg, USA
Friedemann Mattern, Switzerland
John C. Mitchell, USA
Moni Naor, Israel
C. Pandu Rangan, India
Bernhard Steffen, Germany
Demetri Terzopoulos, USA
Doug Tygar, USA
Gerhard Weikum, Germany

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

Subline Series Editors

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

Subline Advisory Board

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

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

Isil Dillig · Jens Palsberg (Eds.)

Verification, Model Checking, and Abstract Interpretation

19th International Conference, VMCAI 2018
Los Angeles, CA, USA, January 7–9, 2018
Proceedings

Springer

Editors
Isil Dillig
University of Texas
Austin, TX
USA

Jens Palsberg
University of California
Los Angeles, CA
USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-73720-1 ISBN 978-3-319-73721-8 (eBook)
https://doi.org/10.1007/978-3-319-73721-8

Library of Congress Control Number: 2017963752

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© Springer International Publishing AG 2018
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

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

Preface

This volume contains the papers presented at VMCAI 2018: the International Conference on Verification, Model Checking, and Abstract Interpretation held during January 7–9, 2018, in Los Angeles, co-located with POPL 2018 (the annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages). Previous meetings were held in Port Jefferson (1997), Pisa (1998), Venice (2002), New York (2003), Venice (2004), Paris (2005), Charleston (2006), Nice (2007), San Francisco (2008), Savannah (2009), Madrid (2010), Austin (2011), Philadelphia (2012), Rome (2013), San Diego (2014), Mumbai (2015), St. Petersburg, Florida (2016), and Paris (2017).

VMCAI provides a forum for researchers from the communities of verification, model checking, and abstract interpretation to present their research and aims to facilitate interaction, cross-fertilization, and advancement of hybrid methods that combine these and related areas. VMCAI topics include: program verification, model checking, abstract interpretation, program synthesis, static analysis, type systems, deductive methods, decision procedures, theorem proving, program certification, debugging techniques, program transformation, optimization, hybrid and cyber-physical systems.

This year the conference received 43 submissions, of which 24 were selected for publication in the proceedings. Each submission was reviewed by at least three Program Committee members, and the main selection criteria were quality, relevance, and originality. In addition to the presentations of the 24 selected papers, the conference also featured an invited tutorial by Mayur Naik (University of Pennsylvania) as well as three invited keynote talks by Ken McMillan (Microsoft Research), Azadeh Farzan (University of Toronto), and Ranjit Jhala (University of California San Diego). We warmly thank them for their participation and contributions.

We would like to thank the members of the Program Committee and the external reviewers for their excellent work. We also thank the members of the Steering Committee, and in particular Lenore Zuck and Andreas Podelski, for their helpful advice, assistance, and support. We thank Annabel Satin for her help in coordinating the events co-located with POPL 2018, and we thank the POPL 2018 Organizing Committee for providing all the logistics for organizing VMCAI. We are also indebted to EasyChair for providing an excellent conference management system.

Finally, we would like to thank our sponsors, Amazon Web Services and Facebook, for their valuable contributions to VMCAI 2018.

November 2017

Isil Dillig
Jens Palsberg

Organization

Program Committee

Jade Alglave	University College London, UK
Roderick Bloem	Graz University of Technology, Austria
Wei-Ngan Chin	National University of Singapore, Singapore
Maria Christakis	MPI-SWS, Germany
Patrick Cousot	New York University, USA
Isil Dillig	The University of Texas at Austin, USA
Laure Gonnord	University of Lyon/Laboratoire d'Informatique du Parallélisme, France
Eric Koskinen	Yale University, USA
Laura Kovacs	Vienna University of Technology, Austria
Paddy Krishnan	Oracle, Australia
Ondrej Lhotak	University of Waterloo, Canada
Ruben Martins	Carnegie Mellon University, USA
Ken McMillan	Microsoft, USA
Daniel Neider	Max Planck Institute for Software Systems, Germany
Jens Palsberg	University of California, Los Angeles, USA
Corina Pasareanu	CMU/NASA Ames Research Center, USA
Andreas Podelski	University of Freiburg, Germany
Xiaokang Qiu	Purdue University, USA
Noam Rinetzky	Tel Aviv University, Israel
Philipp Ruemmer	Uppsala University, Sweden
Roopsha Samanta	Purdue University, USA
Rahul Sharma	Microsoft, India
Ana Sokolova	University of Salzburg, Austria
Tachio Terauchi	Waseda University, Japan
Thomas Wahl	Northeastern University, USA
Thomas Wies	New York University, USA
Charles Zhang	The Hong Kong University of Science and Technology, SAR China

Additional Reviewers

Abdullah, Syed Md Jakaria
An, Shengwei
Antonopoulos, Timos
Bartocci, Ezio
Biere, Armin
Cai, Zhuohong
Costea, Andreea
Darais, David
Ebrahimi, Masoud
Esterle, Lukas
Forget, Julien
Gorogiannis, Nikos

Gotlieb, Arnaud
Gu, Yijia
Guatto, Adrien
Hoenicke, Jochen
Humenberger, Andreas
Jansen, Nils
Kiefer, Stefan
Krishna, Siddharth
Lazic, Marijana
Liu, Peizun
Lozes, Etienne
Mottola, Luca
Moy, Matthieu
Niksic, Filip
Pavlinovic, Zvonimir
Poncelet, Clement
Roeck, Franz
Rusu, Vlad
Schäf, Martin
Stuckey, Peter
Suda, Martin
Udupa, Abhishek
Villard, Jules
Wang, Xinyu

Abstracts of Invited Talks

Rethinking Compositionality for Concurrent Program Proofs

Azadeh Farzan

University of Toronto

Abstract. Classical approaches to reasoning about concurrency are based on reductions to sequential reasoning. Typical tactics are to reason about the global behaviour of the system (commonly employed in model checking) or to reason about the behaviour of each thread independently (such as in Owicki-Gries or Rely/Guarantee). We will discuss a new foundation for reasoning about multi-threaded programs, which breaks from this mold. In the new approach, proof ingredients extracted from a few distinct program behaviours are used as building blocks to a program proof that is free to follow the program control structure when appropriate and break away from it when necessary. Our algorithmic solution to the automated construction of these proofs leverages the power of sequential reasoning similar to the classical techniques, but the sequential reasoning lines need not be drawn at the thread boundaries.

Keywords: Proofs · Concurrency · Compositionality

Reasoning About Functions

Ranjit Jhala

University of California, San Diego

Abstract. SMT solvers' ability to reason about equality, arithmetic, strings, sets and maps, have transformed program analysis and model checking. However, SMT crucially relies on queries being restricted to the above theories which preclude the specification and verification of specification and verification of properties of higher-order, user-defined functions. In this talk, we will describe some recent progress towards removing this restriction by presenting two algorithms for SMT-based reasoning about functions.

The first algorithm, FUSION, enables *abstract* reasoning about functions. FUSION generalizes the first-order notions of strongest post-conditions and summaries to the higher-order setting to automatically synthesize the most precise representation of functions expressible in the SMT logic. Consequently, FUSION yields a relatively complete algorithm for verifying specifications over SMT-decidable theories. While this suffices to verify classical (1-safety) specifications, e.g. array-bounds checking, it does not apply to general (k-safety) specifications over user defined functions, *e.g.* that certain functions are commutative or associative.

The second algorithm, PLE *Proof by Logical Evaluation* (PLE), enables *concrete* reasoning about functions, by showing how to mimic computation within SMT-logics. The key idea is to represent functions in a guarded form and repeatedly unfold function calls under enabled guards. We formalize a notion of an equational proof and show that PLE is complete, *i.e.*is guaranteed to find an equational proof if one exists. Furthermore, we show that PLE corresponds to a universal (or must) abstraction of the concrete semantics of the user-defined functions, and hence, terminates, yielding a precise and predictable means of automatically reasoning about user-defined functions.

Joint work with Benjamin Cosman, Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan Scott, Ryan Newton and Philip Wadler.

How to Stay Decidable

Kenneth McMillan

Microsoft Research

Abstract. Automated provers can substantially increase productivity in the formal verification of complex systems. However, the unpredictability of automated provers presents a major hurdle to usability of these tools. This problem is particularly acute in case of provers that handle undecidable logics, for example, first-order logic with quantifiers.

On the other hand, there is a long history of work on *decidable* logics or fragments of logics. Generally speaking, decision procedures for these logics perform more predictably and fail more transparently than provers for undecidable logics. In particular, in the case of a false proof goal, they usually can provide a concrete counter-model to help diagnose the problem. The problem that remains little studied is how to apply these logics in practice in the proof of large systems. That is, how do we effectively decompose the proof of the system into lemmas couched in decidable fragments, and is the human effort required to do this repaid by more reliable automation?

To answer these questions, we must address the fact that combinations of decidable theories are generally not decidable, and that useful decidable fragments are generally not closed under conjunction. This requires us to practice separation of concerns. For example, it is important to express the implementation of a protocol in a language that captures the protocol's logic without mixing in low-level details such as data structures. Moreover, modularity is an important tool for avoiding undecidability. For example, we can use a high-level protocol model to prove global properties, which are then used as lemmas in proving correctness of the protocol implementation. This allows us to separate invariants which, if combined, would take us outside the decidable realm. In particular, this strategy allows us to produce verification conditions that are decidable because they use function symbols in a stratified way.

Preliminary experience indicates that it is possible to produce verified implementations of distributed protocols in this way with reduced proof complexity and greater reliability of proof automation, without sacrificing execution performance.

Maximum Satisfiability in Program Analysis: Applications and Techniques (Invited Tutorial)

Mayur Naik[1], Xujie Si[1], Xin Zhang[1], and Radu Grigore[2]

[1] University of Pennsylvania
[2] University of Kent

Abstract. A central challenge in program analysis concerns balancing different competing tradeoffs. To address this challenge, we propose an approach based on the Maximum Satisfiability (MaxSAT) problem, an optimization extension of the Boolean Satisfiability (SAT) problem. We demonstrate the approach on three diverse applications that advance the state-of-the-art in balancing tradeoffs in program analysis. Enabling these applications on real-world programs necessitates solving large MaxSAT instances comprising over 10^{30} clauses in a sound and optimal manner. We propose a general framework that scales to such instances by iteratively expanding a subset of clauses while providing soundness and optimality guarantees. We also present new techniques to instantiate and optimize the framework.

Keywords: Maximum satisfiability · Program analysis

Contents

Parameterized Model Checking of Synchronous Distributed Algorithms by Abstraction 1
Benjamin Aminof, Sasha Rubin, Ilina Stoilkovska, Josef Widder, and Florian Zuleger

Gradual Program Verification 25
Johannes Bader, Jonathan Aldrich, and Éric Tanter

Automatic Verification of RMA Programs via Abstraction Extrapolation 47
Cedric Baumann, Andrei Marian Dan, Yuri Meshman, Torsten Hoefler, and Martin Vechev

Scalable Approximation of Quantitative Information Flow in Programs 71
Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf

Code Obfuscation Against Abstract Model Checking Attacks 94
Roberto Bruni, Roberto Giacobazzi, and Roberta Gori

Abstract Code Injection: A Semantic Approach Based on Abstract Non-Interference 116
Samuele Buro and Isabella Mastroeni

A Framework for Computer-Aided Design of Educational Domain Models 138
Eric Butler, Emina Torlak, and Zoran Popović

Automatic Verification of Intermittent Systems 161
Manjeet Dahiya and Sorav Bansal

On abstraction and compositionality for weak-memory linearisability. 183
Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong

From Shapes to Amortized Complexity 205
Tomáš Fiedor, Lukáš Holík, Adam Rogalewicz, Moritz Sinn, Tomáš Vojnar, and Florian Zuleger

Invariant Generation for Multi-Path Loops with Polynomial Assignments. . . . 226
Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács

Analyzing Guarded Protocols: Better Cutoffs, More Systems, More Expressivity 247
Swen Jacobs and Mouhammad Sakr

Refinement Types for Ruby 269
Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak

Modular Analysis of Executables Using On-Demand Heyting Completion 291
Julian Kranz and Axel Simon

Learning to Complement Büchi Automata 313
Yong Li, Andrea Turrini, Lijun Zhang, and Sven Schewe

P^5: Planner-less Proofs of Probabilistic Parameterized Protocols 336
Lenore D. Zuck, Kenneth L. McMillan, and Jordan Torf

Co-Design and Verification of an Available File System 358
Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster

Abstraction-Based Interaction Model for Synthesis 382
Hila Peleg, Shachar Itzhaky, and Sharon Shoham

Generating Tests by Example 406
Hila Peleg, Dan Rasin, and Eran Yahav

A Logical System for Modular Information Flow Verification 430
Adi Prabawa, Mahmudul Faisal Al Ameen, Benedict Lee, and Wei-Ngan Chin

On Constructivity of Galois Connections 452
Francesco Ranzato

Revisiting MITL to Fix Decision Procedures 474
Nima Roohi and Mahesh Viswanathan

Selfless Interpolation for Infinite-State Model Checking 495
Tanja Schindler and Dejan Jovanović

An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs 516
Laura Titolo, Marco A. Feliú, Mariano Moscato, and César A. Muñoz

Author Index 539

Parameterized Model Checking of Synchronous Distributed Algorithms by Abstraction*

Benjamin Aminof[1], Sasha Rubin[2], Ilina Stoilkovska[1](✉), Josef Widder[1], and Florian Zuleger[1]

[1] TU Wien, Vienna, Austria
{benj, stoilkov, widder, zuleger}@forsyte.at
[2] Università degli Studi di Napoli Federico II, Naples, Italy
sasha.rubin@unina.it

Abstract. Parameterized verification of fault-tolerant distributed algorithms has recently gained more and more attention. Most of the existing work considers asynchronous distributed systems (interleaving semantics). However, there exists a considerable distributed computing literature on synchronous fault-tolerant distributed algorithms: conceptually, all processes proceed in lock-step rounds, that is, synchronized steps of all (correct) processes bring the system into the next state.
We introduce an abstraction technique for synchronous fault-tolerant distributed algorithms that reduces parameterized verification of synchronous fault-tolerant distributed algorithms to finite-state model checking of an abstract system. Using the TLC model checker, we automatically verified several algorithms from the literature, some of which were not automatically verified before. Our benchmarks include fault-tolerant algorithms that solve atomic commitment, 2-set agreement, and consensus.

1 Introduction

Fault-tolerant distributed algorithms (FTDAs) are hard to design and prove correct. It is easy to introduce bugs when developing and "optimizing" such distributed algorithms [41]. As we currently see more and more implementations of FTDAs [8, 31, 42, 55], it is desirable to be able to quickly check, whether an optimization did not break the desired behavior. Hence, we observe increasing interest in tool support for eliminating design bugs in distributed algorithms by means of automated verification [1, 2, 17, 18, 26, 28, 35, 45, 52, 54].

The vast majority of the existing literature on verification of distributed systems considers asynchronous systems, that is, the methods are designed for interleaving semantics. Disentangling the methods from the interleaving semantics is challenging. At the same time, there is substantial literature on distributed

* This work is partially supported by the Austrian Science Fund (FWF) via NFN RiSE (S11403, S11405), project PRAVDA (P27722), and the doctoral college LogiCS W1255; and by the Vienna Science and Technology Fund (WWTF) through grant ICT12-059. S. Rubin is a Marie Curie fellow of the Istituto Nazionale di Alta Matematica.

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 1–24, 2018.
https://doi.org/10.1007/978-3-319-73721-8_1

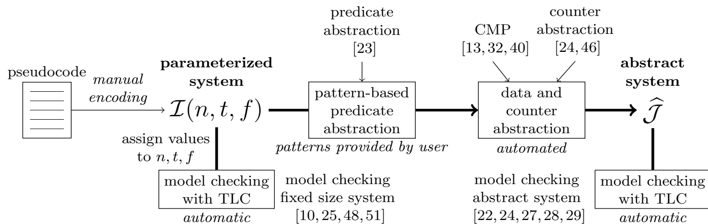


Fig. 1: Overview of our approach and related work

algorithms that are *not* designed for interleaving semantics, namely round-based distributed algorithms [9,11,12,34,37,44,47,53]. In these algorithms, computations proceed in rounds, in which processes perform send, receive and compute transitions in lock-step. There are mainly three reasons for the interest in synchronous distributed algorithms: First, the assumption on synchrony circumvents impossibility results regarding fault-tolerance in asynchronous systems [21]. Second, for hard real-time systems, the underlying hardware and network infrastructure exhibits predictable timing behavior, so that designers of embedded and cyber-physical systems (e.g., in cars and planes) are willing to exploit these timing guarantees at the algorithmic level [30]. Finally, the abstraction of a round that is performed by all processes in lock-step makes it supposedly easier to design algorithms; although there are counterexamples where incorrect synchronous distributed algorithms have been published, as reported in [36].

We focus on verification of synchronous FTDAs, and will adapt and combine several verification methods that were originally designed for asynchronous systems, and apply them in the synchronous setting. Fig. 1 gives an overview of our work together with references of related approaches for the asynchronous case. Our main contribution lies in parameterized verification of FTDAs, that is, we want to verify a distributed algorithm that is executed by n processes in an environment where f processes fail, and designed to work if at most t processes fail, for all values n, t, and f that satisfy some arithmetic conditions, e.g., $f \leq t < n$. This algorithm is formalized as a parameterized system $\mathcal{I}(n,t,f)$, to which we apply abstraction to obtain a finite abstract system $\widehat{\mathcal{J}}$ which serves as input to TLC [50], the model checker associated with TLA+ [33].

To understand the trade-off between parameterized model checking and model checking of fixed size systems, we also did verification of the latter, that is, we fix n, t, and f to small values, e.g., 5, 3, and 2. The resulting fixed size system again serves as input to TLC, as shown in the figure. Our experiments show that model checking fixed sized systems quickly runs into combinatorial state space explosion. This confirms that to verify systems of bigger size, one needs to rely on abstractions, which give verification results for systems of all sizes.

There are several existing approaches for verifying round-based distributed algorithms. Fixed size systems, i.e., (small) instances, were verified using model checking, e.g., in [10, 48, 51]. The following two approaches to parameterized

verification are most related as they also target the round-based model from [12]: [17] proposes invariant checking using decision procedures, requiring the user to provide invariants manually. [38] gives a cut-off theorem for reducing the parameterized problem to verification of small systems (5 to 7 processes). This cut-off theorem considers only consensus algorithms [12], while we are also interested in other algorithms, e.g., k-set agreement or atomic commitment. We discuss the relation of [38] to our work in more detail in Section 6.

Contributions. We introduce a new technique for parameterized model checking of synchronous distributed algorithms.

- We introduce a special *guarded command language* for distributed algorithms, and show that this language allows effective verification by abstraction.
- We combine *automated abstractions* [15,24,32,40,46] that had been introduced for asynchronous systems, and adapt them to synchronous systems.
- Our *modeling framework* uses an independent environment to express the semantics of the code in the presence of faults. While we focus on crash faults in this paper, in the future this will allow us to express semantics of other faults models (e.g., omission, Byzantine) in a modular way.
- We introduce pattern-based *predicate abstraction* for termination guards. This allows verification engineers to specify verification conditions for specific guards and environments. For termination guards found in many synchronous algorithms, we provide verification conditions, which can be reused.
- We do experiments on several synchronous FTDAs [9, 11, 37, 47], some of which were not automatically verified before. Our experiments show that parameterized model checking performs better than checking fixed-size systems already for few (typically 5) processes.

2 Overview on our Approach

A synchronous distributed algorithm runs on a fully connected network of $n \in \mathbb{N}$ processes, which communicate with each other by exchanging messages. The computations are organized in rounds; each round consists of two phases: (1) the *message exchange phase* in which each process broadcasts a message to all others, and (2) the *state transition phase* in which processes update their variables based on the messages received. The processes work synchronously in the sense that they simultaneously switch to the next phase of every round.

We focus on *fault-tolerant agreement* algorithms [5, 37, 47], where processes irrevocably decide a value depending on the initial values of all processes. There are multiple agreement problems in the literature that differ in the way the decision values are related to the initial values. In *consensus*, the processes reach agreement on a value that has been initially proposed by at least one process. In *2-set agreement*, processes may decide on one out of two different values from the set of initial values. In *non-blocking atomic commit*, the processes decide the value *abort* if there is at least one process that initially proposed *abort*, and decide the value *commit* if all processes initially proposed *commit*.

***FloodSet* algorithm:**
The message alphabet consists of subsets of $W = \{0,1\}$.
$v_0 \in W$ is a default decision value
$r \in \mathbb{N}$ is the round number, initially 0
states$_i$:
  $w \subseteq W$, initially containing i's initial value
  $d \in \{0,1,un\}$, initially un
msgs$_i$
1. if $r \leq t$ then
2.   send w to all other processes
3. $r := r+1$
trans$_i$
4. let X_j be the message from j, for each j from which a message arrives
5. $w := w \cup \bigcup_j X_j$
6. if $r = t+1$ then
7.   if $|w| = 1$ then $d := v$, where $w = \{v\}$
8.   else $d := v_0$

(a) The pseudocode of *FloodSet*

Validity: If all processes start with the same initial value, then this is the only possible decision value.
Agreement: No two correct processes decide on different values.
Termination: All correct processes eventually decide.

(b) Specifications

$$s.\mathbf{w} = \begin{bmatrix}\{0\}\\\{0\}\\\{1\}\\\{0\}\\\{0\}\end{bmatrix} \; s.\mathbf{d} = \begin{bmatrix}un\\un\\un\\un\\un\end{bmatrix} \; s.\mathbf{Msg} = \begin{bmatrix}\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\end{bmatrix} \; s.r = 1 \; s.\mathbf{cr} = \begin{bmatrix}\bot\\\bot\\\top\\\bot\\\bot\end{bmatrix} \; s.\mathbf{fld} = \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \; s.\mathbf{rcv} = \begin{bmatrix}\top&\top&\top&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\end{bmatrix}$$

(c) $s \in \mathcal{S}(n,t,f)$, $n = 5, t = 3, f = 2$

$$s'.\mathbf{w} = \begin{bmatrix}\{0\}\\\{0\}\\\{1\}\\\{0\}\\\{0\}\end{bmatrix} \; s'.\mathbf{d} = \begin{bmatrix}un\\un\\un\\un\\un\end{bmatrix} \; s'.\mathbf{Msg} = \begin{bmatrix}\{0\}&\{0\}&\{1\}&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\end{bmatrix} \; s'.r = 1 \; s'.\mathbf{cr} = \begin{bmatrix}\bot\\\bot\\\top\\\bot\\\bot\end{bmatrix} \; s'.\mathbf{fld} = \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} \; s'.\mathbf{rcv} = \begin{bmatrix}\top&\top&\top&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\end{bmatrix}$$

(d) $s' \in \mathcal{S}(n,t,f)$, for $n = 5, t = 3, f = 2$

$$(r = t+1) \wedge (w = \{0\}) \wedge (\forall j\ msg[j] \neq \{1\}) \wedge (\forall j\ msg[j] \neq \{0,1\}) \;\rightarrow\; w := \{0\}, d := 0$$

(e) A guarded assignment defined for *FloodSet*

$$pr \wedge (w = \{0\}) \wedge (\forall j\ msg[j] \neq \{1\}) \wedge (\forall j\ msg[j] \neq \{0,1\}) \;\rightarrow\; w := \{0\}, d := 0$$

(f) Predicate abstraction: the termination guard $(r = t+1)$ is replaced by predicate pr

Fig. 2: The *FloodSet* algorithm

We aim at checking that the algorithms for fault-tolerant agreement satisfy their specifications in the presence of at most t faulty processes, where t satisfies some constraint, e.g., $t < n$. We focus on *crash faults*, exhibited by processes that stop working and cannot restart. As a process can crash in the middle of its execution, it can be the case that it sends a message only to a subset of processes.

We discuss the characteristics of these algorithms by using the *FloodSet* consensus algorithm from Fig. 2a as example. Each process has several variables, e.g., in *FloodSet* each process has the variables d and w. The variable $d \in \{0,1,un\}$ stores the value the process decides on (un refers to the process being undecided), and $w \subseteq W = \{0,1\}$ stores the values the process has seen so far (starting with its own initial value, and the ones received in messages). The processes communicate via messages of a finite message alphabet. In *FloodSet*, the message alphabet is the power set of W, and the message that a process sends is the value of its variable w. In the $(t+1)$-st round, each process decides as follows: if $w = \{v\}$, for some $v \in W$, then $d = v$; otherwise d is assigned a default value v_0. We have to verify that the *FloodSet* algorithm tolerates t process crashes where $t < n$.

2.1 Modeling

We model a distributed algorithm as a transition system that is composed of n processes and an environment. The environment captures the fault model

and the round number. The system obtained in such a way is *parameterized* in the parameters n and t, as well as the parameter f which refers to the actual number of crashed processes during an execution of the algorithm. The processes use the values of the parameters n and t; the parameter f is used only by the environment. We are led to distinguish between f and t, as some of our case studies (early deciding consensus) terminate in $min(f+2, t+1)$ rounds [9,11]. Full definitions of the modeling sketched in this section are found in Section 3.

Processes and environment. To model a process, we define *process variables* and *process functions*. The process variables either store values from a finite domain, or are one-dimensional arrays of size n that store information about the other processes, e.g., the messages received in the previous round. The process functions define the way in which the values of the process variables get updated.

The environment describes how processes behave in the presence of crashes and thus it depends only on the fault model. The environment keeps track of the round number, the crashed processes, and for each crashed process, the subset of processes that receive a message from it in the round in which it crashes. The processes and the environment are defined in more detail in Section 3.1.

Global states. The *(global) states* of the parameterized system contain information about the states of the n processes and the environment. For example, in *FloodSet* (e.g., Fig. 2c), we have a one-dimensional array $s.\mathbf{w}$ of size n that store the sets of values w, a one-dimensional array $s.\mathbf{d}$ that stores the decision values d for every process, a two-dimensional array $s.\mathbf{Msg}$ that stores the messages exchanged by the processes, and environment variables: the round number r, the arrays **cr** and **fld** which store information about process crashes in the current and up to the current round, respectively, and the array **rcv**, where the (i, j)-th cell flags whether process i received a message from process j in the current round. Fig. 2d shows the global state after process 3 crashes and only send a message (i.e., $\{1\}$) to process 1. The parameterized system is formally defined in Section 3.2.

Global transitions. A transition models the following steps: (i) the environment increments the round number, and non-deterministically decides on new crashes and new receiver lists; (ii) every process computes a message, which is delivered by the environment (depending on the values of the environment variables); (iii) every correct process updates its finite domain variables, using a set of *guarded assignments* (described below), and its array variables.

The language of *guarded assignments* that we define is powerful enough to capture constructs that typically occur in synchronous distributed algorithms, such as conditional constructs and iteration over process ids. For instance, one can check if there is a process j from which a message was received in the current and the previous round. This construct is used in *early deciding/stopping* consensus algorithms. The guards that compare the round number against a parameter, which we call *termination guards*, are typically used in synchronous agreement algorithms to check whether a certain round is reached, i.e., whether it is safe for a correct process to make a decision (e.g., line 6 of the pseudocode). The guarded

assignments are formalized in Section 3.1. We introduce guarded assignments in order to perform the abstraction steps syntactically, more details of which can be found in Section 4.3.

A guarded assignment defined for the *FloodSet* algorithm is given in Fig. 2e. It defines the update of the finite domain variables w and d in the case when the current round number is equal to $t + 1$ (that is, when the processes decide). The guard is a conjunction of smaller guards, the first one of which is a termination guard. This guarded assignment models one possible outcome of the execution of the pseudocode between lines 5 and 8. If the set of values of the process is $\{0\}$, and there are no messages sent to that process that contain the value 1, then in the new control state, the set of values remains the same, and the decision value is set to 0. The remaining guarded assignments that model the pseudocode between lines 5 and 8 follow a similar pattern.

2.2 Abstraction

We build a single abstract finite state system, which is not parameterized, and simulates the behavior of every concrete system. Our abstraction is applied in two steps: first t and f are eliminated using *pattern-based predicate abstraction*, and then n is eliminated using *data and counter abstraction.*

Predicate abstraction. The set of guarded assignments defined for the algorithm can contain termination guards that feature the parameter t. For each such guard, we introduce a Boolean predicate, which is true when the guard is satisfied. For every newly defined predicate in this abstraction step, a constraint that ensures that the predicate is eventually satisfied is introduced. This eliminates the parameter t. The parameter f is eliminated by introducing a constraint which states that the faults eventually stop appearing. The predicate abstraction step is described in more detail in Section 4.1.

Fig. 2f shows the guarded assignment from Fig. 2e in which the termination guard $r = t + 1$ is replaced by a Boolean predicate pr. We add the constraint $\mathsf{F}\, pr$ for this predicate, saying that eventually the $(t + 1)$-st round is reached.

Data and counter abstraction. Using ideas from [13,32,40], we fix a small number of processes (two or three), whose behaviors we keep concrete, and abstract the remaining processes depending on the current values of their variables. The choice of the number of fixed processes depends on the properties we are interested in verifying. For the *FloodSet* algorithm, we fix this number to two, as in order to check the agreement property (Fig. 2b), we need to check whether every pair of processes agree on a value. Using data and counter abstraction [24,46], we reduce the size of the array variables in the global state from n to a fixed number, which depends on the number of fixed processes, and the states the remaining processes are in, but is independent of n. The main idea is to store whether there are no processes (*zero*) or at least one process (*many*) that has some particular state. Section 4.2 formally describes the zero-many data and counter abstractions.

Consider the state s' of *FloodSet* in Fig. 2d. We fix processes 1 and 2, and abstract the behavior of processes 3, 4, and 5. Process 3 has a different valuation of the variables w and d than processes 4 and 5, that have the same valuation. Thus, in state s', we say that, e.g., there are *many* processes in the state where $w = \{0\}$ and $d = un$, and there are *zero* processes in the state where $w = \{0, 1\}$ and $d = un$, as there are no processes in s' with these values for the variables.

3 Modeling and Specifications of Synchronous FTDAs

We formalize FTDAs by introducing process variables, process functions, environment variables and parameters n, t, and f. As we consider crash faults, we assume that the parameters satisfy the *resilience condition* $f \leq t < n$. In this section we define a transition system $\mathcal{I}(n, t, f) = \langle \mathcal{S}(n, t, f), \mathcal{S}_0(n, t, f), \mathcal{Q}(n, t, f)\rangle$, called an FTDA instance, for each value of n, t, and f that satisfies the resilience condition.

Notation. A *transition system* is a tuple $M = \langle S, S^0, R\rangle$ where S is a set of *states*, $S^0 \subseteq S$ is a set of *initial states*, and $R \subseteq S \times S$ is a *transition relation*. An *execution* is a path in M that starts in an initial state. Typically, states are valuations of some fixed set of variables X. We write $s.x$ for the value of variable $x \in X$ at state s. For $n \in \mathbb{N}$ we write $[n]$ for the set $\{1, 2, \cdots, n\}$.

3.1 Processes and Environment

A *process* is modeled using *process variables* and *process functions*.

Definition 1 (Process variables). *Let V be a finite set of* process variables*, partitioned into* process control variables $cntl(V) = \{x_1, \cdots, x_{|cntl(V)|}\}$ *and* process neighborhood arrays $nbhd(V) = \{y_1, \cdots, y_{|nbhd(V)|}\}$*. For $v \in V$, let D_v denote the finite set of* values *that v can take if $v \in cntl(V)$, or that each cell in v can take if $v \in nbhd(V)$. We assume that for every $y \in nbhd(V)$, the domain D_y contains a special null value $\perp$ which signifies an* empty cell.

A special neighborhood array is $msg \in nbhd(V)$, which is used to store the messages the process receives in the current round. For convenience, we write M instead of D_{msg}, and call it the message alphabet.

Definition 2 (Process states). *A* process state *p is a valuation of all the variables in V, i.e., an element of $P = \prod_{x \in cntl(V)} D_x \times \prod_{y \in nbhd(V)} (D_y)^n$. We write $p.control$, called a* process control state*, for the valuation restricted to $cntl(V)$. Let $C = \prod_{x \in cntl(V)} D_x$ denote the set of all process control states, and let $C_0 \subseteq C$ denote a set of* initial control states.

We define the domain and range of three *process functions*, the last one being parameterized by n and r (the round number).

Definition 3 (Process functions). *Let F be the set of* process functions $F = \{snd_msg\} \cup \{h_y : y \in nbhd(V) \setminus \{msg\}\} \cup \{update_{n,r} : n, r \in \mathbb{N}\}$*, where*

snd_msg $: C \to M$ *maps process control states to messages;* $h_y : M \to D_y$ *maps messages to values in* D_y *and satisfies the restriction that* $h_y(\perp) = \perp$*; and* $update_{n,r} : P \to C$ *maps process states to control states.*

We use process functions to formally break down and encode the algorithm executed by the processes. Note that the functions *snd_msg* and h_y (for every y) are fixed and finite, whereas $update_{n,r}$ is parameterized by n and r and represents an infinite set of finite functions. This infinite set of functions is defined using a finite set of guarded assignments from the following language.

Each *guarded assignment* is of the form $g \to asg$, where g is a *guard* and *asg* is an *assignment*. An *assignment asg* is a partial function with domain $cntl(V)$ such that if $asg(x)$ is defined then $asg(x) \in D_x$. The *guards* are Boolean combinations (negation and conjunction) of *basic guards*, and are evaluated over process states. The following are the basic guards:

guard	notation		evaluation
empty	$g^{\mathbf{true}}$		**true**
control	$g^{x,v}$	where $x \in cntl(V)$ and $v \in D_x$	$x = v$
termination	$g^{\rhd,\phi(n,t)}$	where $\rhd \in \{>, =\}$, and $\phi(n,t)$ is a linear combination of n and t	$r \rhd \phi(n,t)$
neighborhood	g^{Ψ}	where Ψ is a set of triples $(y, \square, v)$ s.t. $y \in nbhd(V), \square \in \{=, \neq\}$, and $v \in D_y$	$\exists j \in [n]$ $\bigwedge_{\Psi}(y[j] \square v)$

We write $p \models g$ to signify that process state p satisfies the guard g.

Given a guarded assignment $g \to asg$ and parameters n, r, we define the induced function $update_{n,r}$ as follows. Let $p \in P$ be a process state. If $p \not\models g$ then $update_{n,r}(p) = p.control$. If $p \models g$ then $update_{n,r}(p) = c$ where $c.x = p.x$ if $asg(x)$ not defined, and $c.x = asg(x)$ otherwise.

To fully characterize the function $update_{n,r}$, we associate with it a finite set G of guarded assignments, where the guards are pairwise mutually exclusive.

The guards capture various constructs found in the distributed computing literature. For example, the empty guard captures simple assignments, Boolean combinations of control and termination guards capture conditionals, and Boolean combinations of the neighborhood guards capture iteration over process ids when traversing the process neighborhood arrays.

Since the process functions serve as the building blocks of the transition relation (as we formally explain later), in Section 4 where we abstract the transition system, we will also have to abstract the process functions. Towards this end, a key step will involve abstracting the guarded assignments. This step is done syntactically, by defining abstract versions of the basic guards.

Definition 4 (Environment variables V^e). *The environment variables are:* r*, with domain* $D^e_r = \mathbb{N}$*, is the current round number;* **cr***, with domain* $D^e_{\mathbf{cr}} = \{\perp, \top\}^n$*, flags the crashed processes in the current round, with the value* $\top$ *indicating a crash;* **fld***, with domain* $D^e_{\mathbf{fld}} = \{\perp, \top\}^n$*, flags the processes that crashed in some previous round; and* **rcv***, with domain* $D^e_{\mathbf{rcv}} = \{\perp, \top\}^{n \cdot n}$*, stores a receivers list for every process, that defines the subset of processes to which the*

process sends a message in the current round, with the value $\top$ *in the* (i, j)*-th cell indicating that process* i *receives the message from process* j*.*

3.2 FTDA Instance $\mathcal{I}(n, t, f)$

We define the transition system $\mathcal{I}(n, t, f) = \langle \mathcal{S}(n, t, f), \mathcal{S}_0(n, t, f), \mathcal{Q}(n, t, f) \rangle$, as a combination of n processes and the environment, as follows.

Global states $\mathcal{S}(n, t, f)$*.* The set $\mathcal{S}(n, t, f)$ of *global states* of an FTDA instance is the set of all possible valuations of the following *FTDA variables* $\mathcal{V}$:

Definition 5 (FTDA variables). *The set* $\mathcal{V}$ *is the union of the sets of:*

- control variables $cntl(\mathcal{V})$*, containing one-dimensional array variables* $\mathbf{x}$ *of size* n*, that range over* $(D_x)^n$*, where* $x \in cntl(V)$ *is a process control variable.*
- neighborhood arrays $nbhd(\mathcal{V})$*, containing two-dimensional array variables* $\mathbf{Y}$ *of size* $n \times n$*, ranging over* $(D_y)^{n \cdot n}$*, with* $y \in nbhd(V)$ *a process neighborhood array. The neighborhood array corresponding to the process neighborhood array msg is denoted* **Msg***, and is called the* message channel.
- *environment variables* V^e*.*

Intuitively, the variables in $cntl(\mathcal{V}) \uplus nbhd(\mathcal{V})$ are used to store the values of the process variables of each of the n processes in the FTDA instance, and the value of $\mathbf{Msg}[i, j]$ is equal to the value of $msg[j]$ of process i.

To define the rest of the FTDA instance, we need the following notations. For a global state s and $i \in [n]$, we denote by:

- $s.control_i$ the tuple $\langle s.\mathbf{x}_1[i], \dots, s.\mathbf{x}_{|cntl(\mathcal{V})|}[i] \rangle \in C$;
- $s.row_i^{\mathbf{Y}}$ the tuple $\langle s.\mathbf{Y}[i, 1], \dots, s.\mathbf{Y}[i, n] \rangle \in (D_y)^n$ (for $\mathbf{Y} \in nbhd(\mathcal{V})$);
- $s.local_i$ the tuple $\langle s.control_i, s.row_i^{\mathbf{Y}_1}, \dots, s.row_i^{\mathbf{Y}_{|nbhd(\mathcal{V})|}} \rangle \in P$.
- $s.location_i$ the tuple $\langle s.control_i, s.\mathbf{fld}[i] \rangle \in Loc$, where $Loc = C \times \{\bot, \top\}$ is the set of *process locations.*

A *process location* is a pair of the process control state and a failure flag fld, whose value is stored in the environment variable **fld**. As we will see in Section 4, we need the notion of process location in our abstractions, as we need to distinguish between correct and crashed processes that are in the same control state.

Initial global states $\mathcal{S}_0(n, t, f)$*.* A global state s is *initial* if the values it assigns to the different variables satisfy the following restrictions: the values of the control variables are initial, i.e., $s.control_i \in C_0$ for every $i \in [n]$ (recall that C_0 is the set of initial control states); all the cells of all the neighborhood arrays are empty, i.e., $s.\mathbf{Y}[i, j] = \bot$ for all $i, j \in [n]$ and all $\mathbf{Y} \in nbhd(\mathcal{V})$; and the environment variables are initialized as follows: (i) $s.r = 0$, (ii) $s.\mathbf{cr}[i] = \bot$, for all $i \in [n]$, (iii) $s.\mathbf{fld}[i] = \bot$, for all $i \in [n]$, and (iv) $s.\mathbf{rcv}[i, j] = \bot$, for all $i, j \in [n]$.

Transition relation $\mathcal{Q}(n,t,f)$. We define three transition relations: $\xrightarrow{\mathsf{ENV}}$ updates the environment variables; $\xrightarrow{\mathsf{MEP}}$ captures the message exchange phase; $\xrightarrow{\mathsf{PROC}}$ updates the control variables and neighborhood arrays. A transition of the FTDA instance is an element of the composition $\xrightarrow{\mathsf{ENV}}\xrightarrow{\mathsf{MEP}}\xrightarrow{\mathsf{PROC}}$, i.e., $(s, s''') \in \mathcal{Q}(n,t,f)$ iff there exist states $s', s'' \in \mathcal{S}(n,t,f)$ such that $s \xrightarrow{\mathsf{ENV}} s' \xrightarrow{\mathsf{MEP}} s'' \xrightarrow{\mathsf{PROC}} s'''$.

Updating environment variables. We define $s \xrightarrow{\mathsf{ENV}} s'$ as follows. First, the round number is incremented, i.e. $s'.r = s.r + 1$.

Second, the environment chooses which processes will crash in the current round, while keeping the number of crashed processes below the parameter f. That is, $s'.\mathbf{cr}$ is updated to a value that satisfies the following conditions: (i) for every $i \in [n]$, we have $s'.\mathbf{cr}[i] = \bot$ if $s.\mathbf{fld}[i] = \top$, and (ii) $|\{i \in [n] \mid s.\mathbf{fld}[i] \vee s'.\mathbf{cr}[i]\}| \leq f$. Intuitively, condition (ii) reflects the non-deterministic assignment of values to $\mathbf{cr}$, by allowing at most f processes to be flagged as crashed in every execution.

Finally, the receiver lists for the next round are updated by flagging that no message is received from processes that crashed in some previous round, receiving all messages from the correct processes, and non-deterministically choosing which processes receive messages from the processes that crash in the current round. That is, for every $i, j \in [n]$, the following holds for $s'.\mathbf{rcv}$: (i) if $s.\mathbf{fld}[j] = \top$ then $s'.\mathbf{rcv}[i,j] = \bot$, and (ii) if $s.\mathbf{fld}[j] = \bot$ and $s'.\mathbf{cr}[j] = \bot$, then $s'.\mathbf{rcv}[i,j] = \top$.

Message exchange phase. In this transition, the cell (i, j) of the message channel is assigned the message sent from process j to process i, if i is in the receiver list of j for this round. We define $s \xrightarrow{\mathsf{MEP}} s'$ if (i) $s'.\mathbf{Msg}[i,j] = \mathit{snd_msg}(s.\mathit{control}_j)$ if $\mathbf{rcv}[i,j] = \top$, and (ii) $s'.\mathbf{Msg}[i,j] = \bot$ if $\mathbf{rcv}[i,j] \neq \top$.

Updating process variables. In this transition, the failure flags are updated, and every correct process first applies the process function $\mathit{update}_{n,r}$ to update its control variables, and then updates its neighborhood arrays (except for msg) using the messages it received.

We define $s \xrightarrow{\mathsf{PROC}} s'$ as follows. First, the failure flags are updated, i.e., for all $i \in [n]$, $s'.\mathbf{fld}[i] = s.\mathbf{fld}[i] \vee s.\mathbf{cr}[i]$. Second, the control variables are updated as follows: (i) for all $i \in [n]$, $s'.\mathit{control}_i = \mathit{update}_{n,s.r}(s.\mathit{local}_i)$ if $s'.\mathbf{fld}[i] = \bot$; and (ii) $s'.\mathit{control}_i = s.\mathit{control}_i$ otherwise. Third, the neighborhood arrays are updated as follows: for every $i \in [n]$ and every $\mathbf{Y} \in \mathit{nbhd}(\mathcal{V}) \setminus \{\mathbf{Msg}\}$: (i) $s'.\mathbf{Y}[i,j] = h_y(s.\mathbf{Msg}[i,j])$, for all $j \in [n]$, if $s'.\mathbf{fld}[i] = \bot$, and (ii) $s'.\mathbf{Y}[i,j] = s.\mathbf{Y}[i,j]$, for all $j \in [n]$, otherwise. Finally, the the message channel is flushed, i.e., $s'.\mathbf{Msg}[i,j] = \bot$, for every $i, j \in [n]$.

Definition 6 (FTDA instance). *Given process variables V, process functions F, environment variables V^e, and parameter values $n, t, f \in \mathbb{N}$, such that $f \leq t < n$, we define the* FTDA instance $\mathcal{I}(n,t,f)$ *to be the transition system $\langle \mathcal{S}(n,t,f), \mathcal{S}_0(n,t,f), \mathcal{Q}(n,t,f) \rangle$.*

3.3 Specification Language

We use a fragment of indexed linear temporal logic [7, 19] to encode the specifications of distributed algorithms. We define its semantics w.r.t. n processes and a set of Boolean predicates Pred. The state of each process is given by the valuations of a set of variables Vars, where each variable $z \in \mathsf{Vars}$ has an associated domain D_z. A global state ς is given by valuations $\varsigma.\mathbf{z}[i]$ for each variable z and process i and a truth-value assignment $\varsigma.q \in \{\top, \bot\}$ for each $q \in \mathsf{Pred}$. We consider atomic propositions of the form $([z = v], i)$, where $z \in \mathsf{Vars}$, $v \in D_z$ and i is an index, and predicates $q \in \mathsf{Pred}$. We use the following fragment of indexed-LTL:

Definition 7 (The fragment $\mathcal{F}_l$). *For $l \in \mathbb{N}$, we write $\mathcal{F}_l$ for the set of indexed-LTL formulas of the form $\forall i_1 \forall i_2 \cdots \forall i_l.\psi$, where (1) ψ only contains $\exists$-quantifiers, and (2) an $\exists$-quantifier only appears in subformulas of the form $\exists i.([z = v], i)$.*

The semantics of an atomic proposition in a state ς is defined by $\varsigma \models ([z = v], i)$ iff $\varsigma.\mathbf{z}[i] = v$. We define the semantics of a predicate $q \in \mathsf{Pred}$ as follows: $\varsigma \models q$ iff $\varsigma.q = \top$. The semantics of the logical connectives, quantifiers and temporal operators is standard. We will also write $\mathbf{z}[i] = v$ instead of $([z = v], i)$.

The fragments $\mathcal{F}_l$, for $l \in \mathbb{N}$, are rich enough to capture specifications of fault-tolerant agreement. To express specifications of FTDA instances, we define $\mathsf{Vars} = \mathit{cntl}(V) \cup \{\mathit{cr}, \mathit{fld}\}$, where $\mathit{cr}, \mathit{fld}$ are the flags stored in the environment variables $\mathbf{cr}, \mathbf{fld}$. Below, we formalize the specifications stated in Fig. 2b, which are evaluated over global states $s \in \mathcal{S}(n, t, f)$:

- **Validity:** If there is no process with an initial value different from 0, then 0 is the only decision value (there is a symmetric specification for $w = \{1\}$):

$$\forall i.\ (\exists j.\mathbf{w}[j] \neq \{0\}) \vee \mathsf{G}((\mathbf{fld}[i] = \bot \wedge \mathbf{d}[i] \neq \mathit{un}) \rightarrow \mathbf{d}[i] = 0)$$

- **Agreement:** No two correct processes decide differently:

$$\forall i \forall j.\ \mathsf{G}((\mathbf{fld}[i] = \bot \wedge \mathbf{fld}[j] = \bot \wedge \mathbf{d}[i] \neq \mathit{un} \wedge \mathbf{d}[j] \neq \mathit{un}) \rightarrow ((\mathbf{d}[i] = 0 \wedge \mathbf{d}[j] = 0) \vee (\mathbf{d}[i] = 1 \wedge \mathbf{d}[j] = 1)))$$

- **Termination**: Every correct process eventually decides:

$$\forall i.\ \mathsf{F}(\mathbf{fld}[i] = \bot \rightarrow \mathbf{d}[i] \neq \mathit{un})$$

Note that the above stated formulas do not use Boolean propositions, i.e., $\mathsf{Pred} = \emptyset$ (also the global states of FDTA instances do not contain Boolean variables). In Section 4.1, we will state formulas that use Boolean propositions and define abstract transition systems that have Boolean variables.

3.4 Parameterized Model Checking

The *parameterized model checking question* is to decide, given process variables V, process functions F, environment variables V^e, and a specification $\varphi \in \mathcal{F}_l$, whether φ is satisfied in every FTDA instance $\mathcal{I}(n, t, f)$ such that $f \leq t < n$.

3.5 Symmetry

We observe that the FTDA instance $\mathcal{I}(n,t,f)$ is a symmetric transition system [20]. Due to the symmetry, we can fix a small number m of processes that represent any m processes among the n processes of the FTDA instance. To determine m, we take the maximal number of $\forall$-quantifiers that appear in the specifications expressed in our fragment of indexed-LTL. For example, the validity and termination consensus specifications (cf. Section 3.3) have a single $\forall$-quantifier, while agreement has two. Therefore, for consensus we set $m = 2$.

Once we fix m, for every indexed-LTL formula φ, where the indices range over $[n]$, we define a formula φ^m, where the indices bound by $\forall$-quantifiers range over $[m]$, and the indices bound by $\exists$-quantifiers range over $[n]$. We denote by $\mathcal{F}_l^m$ the set of indexed-LTL formulas $\{\varphi^m \mid \varphi \in \mathcal{F}_l \text{ and } l \le m\}$.

Proposition 1 (Symmetry). *The indexed-LTL specification φ is satisfied in an FTDA instance $\mathcal{I}(n,t,f)$ if φ^m is satisfied in $\mathcal{I}(n,t,f)$.*

4 Abstracting Synchronous FTDAs

We define the pattern-based predicate abstraction in Section 4.1, and the zero-many data and counter abstraction in Section 4.2. As these definitions are not effective, in Section 4.3 we give an effective method for processes defined by the process functions in Section 3.1. The challenge lies in the fact that we need to abstract a family of systems parameterized by n, t, and f.

4.1 Predicate Abstraction: Eliminating t and f

Recall that the parameter f refers to the actual number of processes that crash during a run of an instance $\mathcal{I}(n,t,f)$, and the parameter t appears in the termination guards. To build a system $\mathcal{J}(n)$ parameterized only in n, we introduce predicates that abstract basic termination guards, and verification conditions that have to be satisfied in every execution of $\mathcal{J}(n)$.

Predicates. We introduce k Boolean predicates $pr_1, \dots, pr_k$, where k is the number of basic termination guards of the form $r \rhd \phi(n,t)$, appearing in the set G of guarded assignments that define the function $update_{n,r}$. Each predicate pr_j, for $j \in [k]$, is true whenever the basic termination guard it replaces is satisfied. By replacing the basic termination guards with predicates in $\mathcal{J}(n)$, we eliminate the variable variable r from environment, as in $\mathcal{I}(n,t,f)$, the variable r occurs only in the basic termination guards. Thus, the set $\mathcal{V}$ of variables of $\mathcal{J}(n)$ contains:

- control variables $\mathbf{x} \in cntl(\mathcal{V})$, ranging over $(D_x)^n$;
- neighborhood arrays $\mathbf{Y} \in nbhd(\mathcal{V})$, ranging over $(D_y)^{n \cdot n}$;
- environment variables $\mathbf{cr}, \mathbf{fld}, \mathbf{rcv}$.

Additionally, for $\mathcal{J}(n)$, we introduce a set $\mathsf{Pred} = \{pr_1, \dots, pr_k\}$ of Boolean predicates. The set $\Sigma(n)$ of states of $\mathcal{J}(n)$ is the set of all valuations of $\mathcal{V}$. The following abstraction mapping α maps states of $\mathcal{I}(n,t,f)$ to states from $\mathcal{J}(n)$.

Definition 8 (Abstraction mapping α). *We define the abstraction mapping* $\alpha : \mathcal{S}(n,t,f) \to \Sigma(n)$ *as:* $\sigma.\mathbf{x} = s.\mathbf{x}$, *for all* $\mathbf{x} \in \mathit{cntl}(\mathcal{V})$, $\sigma.\mathbf{Y} = s.\mathbf{Y}$, *for all* $\mathbf{Y} \in \mathit{nbhd}(\mathcal{V})$, $\sigma.\mathbf{cr} = s.\mathbf{cr}$, $\sigma.\mathbf{fld} = s.\mathbf{fld}$, $\sigma.\mathbf{rcv} = s.\mathbf{rcv}$ *and for every* $j \in [k]$, $\sigma.pr_j = \top$, *if the basic termination guard* $r \rhd_j \phi_j(n,t)$ *is satisfied in the state* s, *and* $\sigma.pr_j = \bot$, *otherwise.*

Pattern-based verification conditions. We define a set of indexed-LTL formulas $\mathcal{C}$ of *verification conditions* that we impose on $\mathcal{J}(n)$. The formulas in $\mathcal{C}$ introduce restrictions on how the predicates in pr and the crash flags in $\mathbf{cr}$ are assigned values in $\mathcal{J}(n)$ in a way that reflect behaviors of the concrete executions. Note that these verification conditions are imposed on the environment, and can therefore be reused across algorithms that operate under the same environment.

Let *clean* denote the formula $\neg(\exists i\ \mathbf{cr}[i] = \top)$. The formula *clean* is satisfied in a state $\sigma \in \Sigma(n)$, if there is no process that has been flagged as newly crashed in σ. We list the conditions that we identified in multiple benchmarks, together with an explanation of why they hold.

- **$\mathsf{FG}\,\boldsymbol{clean}$** ensures that from some time on, there are no more crashes. It holds because f is finite in each instance.
- **$\mathsf{F}\,(\boldsymbol{pr_j})$ and $\mathsf{FG}\,(\boldsymbol{pr_j})$** for $j \in [k]$ where $\rhd_j$ is $=$ and $\rhd_j$ is $>$, respectively. For instance, the termination guard of *FloodSet* in Fig. 2 is $r = t+1$, and evaluates to true once (in round $t+1$), while a guard $r > t$ becomes and stays true.
- **$(\bigwedge_j \neg \boldsymbol{pr_j})\mathsf{U}\boldsymbol{clean}$** ensures that the basic termination guards become true only after a clean round has occurred. This is typical for consensus algorithms that use a guard $r = t+1$ and are designed for $f \le t$ faults. In this case, in at least one of the $t+1$ rounds no process crashes, i.e., the round is clean.

While currently we perform this abstraction step manually, in the future we aim at developing an automatic procedure for such verification conditions.

Given $\mathcal{I}(n,t,f)$ and $\Sigma(n)$, let $\mathcal{J}(n)$ be the *overapproximation* [14] of $\mathcal{I}(n,t,f)$ induced by α. Let $\mathcal{C}$ be a set of constraints that are satisfied in all instances $\mathcal{I}(n,t,f)$, for $f \le t < n$ (e.g., for *FloodSet*, $\mathcal{C} = \{\mathsf{FG}\,\mathit{clean}, \mathsf{F}\,\mathit{pr}, \mathit{pr}\mathsf{U}\mathit{clean}\}$). Let $\chi_\mathcal{C}$ be the conjunction of all formulas in $\mathcal{C}$. As there are no $\forall$-quantifiers in $\chi_\mathcal{C}$, the formula $\chi_\mathcal{C} \to \varphi^m$ is also in $\mathcal{F}^m_l$.

Proposition 2 (Soundness of α). *For every* $n \in \mathbb{N}$, *if* $\mathcal{J}(n) \models \chi_\mathcal{C} \to \varphi^m$ *then* $\mathcal{I}(n,t,f) \models \varphi^m$, *for all* $t, f \in \mathbb{N}$ *s.t.* $f \le t < n$.

4.2 Zero-many Data and Counter Abstraction

The system $\mathcal{J}(n)$ obtained after applying predicate abstraction is still parameterized in n, i.e., the size of its array variables depends on n. To build a finite system independent of n, we fix the size of the array variables. We proceed by fixing m processes and abstracting the remaining $n-m$ processes based on their process location (defined in Section 3.2). That is, for all process locations $\ell \in \mathit{Loc}$, we store whether no process from the $n-m$ processes is in location ℓ (zero), or whether at least one process from the $n-m$ processes is in location ℓ (many).

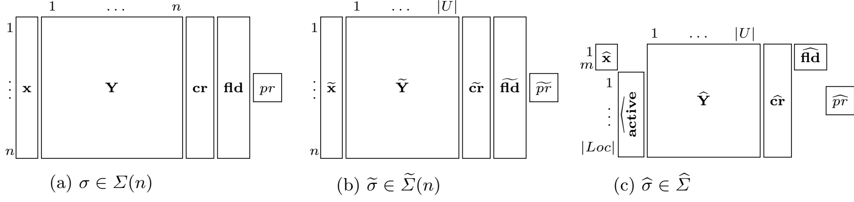


Fig. 3: Two-step zero-many abstraction (for $|cntl(\mathcal{V})| = |nbhd(\mathcal{V})| = 1$, and omitting **rcv** for space reasons) with m fixed processes. **(a)** illustrates a state σ of $\mathcal{J}(n)$; **(b)** shows the result of applying $\widetilde{\alpha}_n$, where $U = [m] \cup Loc$. $\widetilde{\mathbf{Y}}[i, v]$ stores the set of values in row i of $\mathbf{Y}$ for all columns j witnessed by v; **(c)** shows the result of applying $\widehat{\alpha}_n$, i.e., $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{fld}}$ store the control variable x and the failure flag of the m processes, $\widehat{\mathbf{active}}[u]$ stores if there are zero or many processes witnessed by $u \in Loc$, $\widehat{\mathbf{Y}}[u, v]$ stores the union of values $\widetilde{\mathbf{Y}}[i, v]$ for processes i witnessed by u, and $\widehat{\mathbf{cr}}[u]$ stores the union of $\widetilde{\mathbf{cr}}[i]$ for processes i witnessed by u.

We apply the zero-many abstractions in two steps (Fig. 3). After the first step, the overapproximation still depends on n, but after the second step the overapproximation is finite and independent of n.

Step one: Zero-many data abstraction. In the first step, we fix the size of the two-dimensional arrays from the set $nbhd(\mathcal{V}) \cup \{\mathbf{rcv}\}$ so that the number of their columns depends on Loc and m, and not on n. As a result, we obtain an abstract system $\widetilde{\mathcal{J}}(n)$, that is still parameterized in n.

To build $\widetilde{\mathcal{J}}(n)$, we need the following notation. First, we define a set $U = [m] \cup Loc$ of *indices*, which will be used as indices of the abstract array variables. We say that an index $u \in U$ *witnesses* a process, if it corresponds to one of the fixed m processes, i.e., if $u \in [m]$, or if $u \in Loc$ and there exists a process whose current location is u. In this step, we use the elements of U as indices for the columns of the two-dimensional arrays: each cell in a column indexed by $u \in U$ is a union of the cells in the column indexed by the processes witnessed by u.

Next, we define a mapping $ids : \Sigma(n) \times U \to 2^{[n]}$, that maps a state $\sigma \in \Sigma(n)$ and an index $u \in U$ to the set of processes witnessed by u, i.e., $ids(\sigma, u) = \{u\}$, if $u \in [m]$, and $ids(\sigma, u) = \{i \in [n] \setminus [m] \mid \sigma.location_i = u\}$, if $u \in Loc$.

Finally, we define the set of variables $\widetilde{\mathcal{V}}$ of $\widetilde{\mathcal{J}}(n)$, that contains (i) the control variables $\widetilde{\mathbf{x}} \in cntl(\widetilde{\mathcal{V}})$, ranging over $(D_x)^n$, (ii) the neighborhood arrays $\widetilde{\mathbf{Y}} \in nbhd(\widetilde{\mathcal{V}})$, ranging over $(2^{D_y})^{n \cdot |U|}$, and (iii) the environment variables $\widetilde{\mathbf{cr}}, \widetilde{\mathbf{fld}}, \widetilde{\mathbf{rcv}}$, ranging over $\{\bot, \top\}^n$,$\{\bot, \top\}^n$, and $(2^{\{\bot,\top\}})^{n \cdot |U|}$ respectively. The set $\widetilde{\Sigma}(n)$ of states of $\widetilde{\mathcal{J}}(n)$ is the set of all valuations of $\widetilde{\mathcal{V}}$.

Using this notation, we introduce the abstraction mapping $\widetilde{\alpha}_n : \Sigma(n) \to \widetilde{\Sigma}(n)$, that maps states of $\mathcal{J}(n)$ (Fig. 3a) to states from $\widetilde{\Sigma}(n)$ (Fig. 3b).

Definition 9 (Abstraction Mapping $\widetilde{\alpha}_n$). *We define the abstraction mapping* $\widetilde{\alpha}_n : \Sigma(n) \to \widetilde{\Sigma}(n)$ *as:* $\widetilde{\sigma}.\widetilde{\mathbf{x}} = \sigma.\mathbf{x}$*, for all* $\widetilde{\mathbf{x}} \in cntl(\widetilde{\mathcal{V}})$*;* $\widetilde{\sigma}.\widetilde{\mathbf{cr}} = \sigma.\mathbf{cr}$*;* $\widetilde{\sigma}.\widetilde{\mathbf{fld}} = \sigma.\mathbf{fld}$*;*

$\widetilde{\sigma}.\widetilde{pr}_j = \sigma.pr_j$, *for* $j \in [k]$; *and for all* $\widetilde{\mathbf{Y}} \in nbhd(\widetilde{\mathcal{V}}) \cup \{\widetilde{\mathbf{rcv}}\}$, $i \in [n], v \in U$, $\widetilde{\sigma}.\widetilde{\mathbf{Y}}[i,v] = \bigcup\{\sigma.\mathbf{Y}[i,j] \mid j \in ids(\sigma, v)\}$.

Given the system $\mathcal{J}(n)$ and the set $\widetilde{\Sigma}(n)$ of states, we define $\widetilde{\mathcal{J}}(n)$ as the *overapproximation* of $\mathcal{J}(n)$ induced by $\widetilde{\alpha}_n$.

Proposition 3 (Soundness of $\widetilde{\alpha}_n$). *For every* $n \in \mathbb{N}$, *and a formula* $\psi^m \in \mathcal{F}_l^m$, *we have that if* $\widetilde{\mathcal{J}}(n) \models \psi^m$, *then* $\mathcal{J}(n) \models \psi^m$.

Step two: Zero-many counter abstraction In this step, we store the values of the control variables and the failure flags for the m processes in the variables $\widehat{\mathbf{x}} \in cntl(\widehat{\mathcal{V}})$ and $\widehat{\mathbf{fld}}$ respectively, and for the remaining $n - m$ processes, we keep information whether there exists some process from $[n] \setminus [m]$ in some location $\ell \in Loc$ in a newly introduced variable $\widehat{\mathbf{active}}$. Finally, we use the elements from the set U to index the rows of the two-dimensional arrays from the set $nbhd(\widehat{\mathcal{V}}) \cup \{\widehat{\mathbf{rcv}}\}$ and the one-dimensional array $\widehat{\mathbf{cr}}$. Note that the failure flags of the $n - m$ processes are encoded in the process locations. This results in a finite system $\widehat{\mathcal{J}}$, which is not parameterized.

To build $\widehat{\mathcal{J}}$, we first define the mapping $\widetilde{ids} : \widetilde{\Sigma}(n) \times U \to 2^{[n]}$ analogously to the mapping ids above: $\widetilde{ids}(\widetilde{\sigma}, u) = \{u\}$ if $u \in [m]$, and $\widetilde{ids}(\widetilde{\sigma}, u) = \{i \in [n] \setminus [m] \mid \widetilde{\sigma}.location_i = u\}$ otherwise. We define the set $\widehat{\mathcal{V}}$ of variables of $\widehat{\mathcal{J}}$, that contains:

- control variables $\widehat{\mathbf{x}} \in cntl(\widehat{\mathcal{V}})$ of the m fixed processes, ranging over $(D_x)^m$
- the array $\widehat{\mathbf{active}}$, ranging over $\{0, \mathsf{many}\}^{|Loc|}$, that stores for a location $u \in Loc$, whether there are no processes in location u ($\widehat{\mathbf{active}}[u] = 0$), or if there is at least one process in location u ($\widehat{\mathbf{active}}[u] = \mathsf{many}$);
- neighborhood arrays $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}})$, ranging over $(2^{D_y})^{|U| \cdot |U|}$, and
- environment variables $\widehat{\mathbf{cr}}, \widehat{\mathbf{fld}}, \widehat{\mathbf{rcv}}$, ranging over $(2^{\{\bot, \top\}})^{|U|}$, $\{\bot, \top\}^m$, and $(2^{\{\bot, \top\}})^{|U| \cdot |U|}$ respectively.

Using the notation defined above, we define the abstraction mapping $\widehat{\alpha}_n$ that maps an abstract state $\widetilde{\sigma} \in \widetilde{\Sigma}(n)$ (Fig. 3b) to an abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ (Fig. 3c).

Definition 10 (Abstraction mapping $\widehat{\alpha}_n$). *We define the abstraction mapping* $\widehat{\alpha}_n : \widetilde{\Sigma}(n) \to \widehat{\Sigma}$ *as: for* $u \in [m]$, $\widehat{\sigma}.\widehat{\mathbf{x}}[u] = \widetilde{\sigma}.\widetilde{\mathbf{x}}[u]$, *for all* $\widehat{\mathbf{x}} \in cntl(\widehat{\mathcal{V}})$, *and* $\widehat{\sigma}.\widehat{\mathbf{fld}}[u] = \widetilde{\sigma}.\widetilde{\mathbf{fld}}[u]$; *for* $u \in Loc$, $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = 0$ *if* $\widetilde{ids}(\widetilde{\sigma}, u) = \emptyset$, *and* $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$ *otherwise; for* $u \in U$, $\widehat{\sigma}.\widehat{\mathbf{cr}}[u] = \bigcup\{\widetilde{\sigma}.\widetilde{\mathbf{cr}}[i] \mid i \in \widetilde{ids}(\widetilde{\sigma}, u)\}$; *for* $j \in [k]$, $\widehat{\sigma}.\widehat{pr}_j = \widetilde{\sigma}.\widetilde{pr}_j$; *and for all* $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}}) \cup \{\widehat{\mathbf{rcv}}\}$, $u, v \in U$, $\widehat{\sigma}.\widehat{\mathbf{Y}}[u,v]$ *is* $\bigcup\{\widetilde{\sigma}.\widetilde{\mathbf{Y}}[i,v] \mid i \in \widetilde{ids}(\widetilde{\sigma}, u)\}$.

Given $\widetilde{\mathcal{J}}(n)$ and $\widehat{\Sigma}$, we define the abstract system $\widehat{\mathcal{J}}$ as the overapproximation induced by the mapping $\widetilde{\alpha}_n$.

We now define how to evaluate formulas $\psi^m \in \mathcal{F}_l^m$ in states $\widehat{\sigma} \in \widehat{\Sigma}$. As we have removed the parameter n, when evaluating ψ^m in $\widehat{\sigma}$, the indices bound by

the $\exists$-quantifier range over the set U of abstract indices, while the indices bound by the $\forall$-quantifier continue to range over the set $[m]$.

Recall that to express specifications of agreement algorithms we defined $\mathsf{Vars} = \mathit{cntl}(V) \cup \{cr, fld\}$, and atomic propositions in a formula of the form $([z = v], i)$ for $z \in \mathsf{Vars}$, $v \in D_z$ and index i. We now define the meaning of the indexed atomic propositions in $\widehat{\sigma} \in \widehat{\Sigma}$, by distinguishing the following cases:

$z \neq cr$. We define $\widehat{\sigma} \models ([z = v], i)$ if (a) $i \in [m]$ and $\widehat{\sigma}.\widehat{\mathbf{z}}[i] = v$; or (b) $i \in Loc$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[i] = \mathsf{many} \wedge i.z = v$;

$z = cr$. We define $\widehat{\sigma} \models ([cr = v], i)$ if $v \in \widehat{\sigma}.\widehat{\mathbf{cr}}[i]$.

Proposition 4 (Soundness of $\widehat{\alpha}_n$). *For every $n \in \mathbb{N}$, and a formula $\psi^m \in \mathcal{F}_l^m$ we have that if $\widehat{\mathcal{J}} \models \psi^m$ then $\widetilde{\mathcal{J}}(n) \models \psi^m$.*

The overall soundness of our approach is a consequence of Propositions 1 – 4.

Theorem 1 (Soundness). *Let $\mathcal{I}(n, t, f)$ be an FTDA instance, and $\widehat{\mathcal{J}}$ the abstract system defined as the overapproximation induced by the abstraction mapping $\widehat{\alpha}_n \circ \widetilde{\alpha}_n \circ \alpha$. If $\widehat{\mathcal{J}} \models \chi_{\mathcal{C}} \to \varphi^m$, then $\mathcal{I}(n, t, f) \models \varphi$.*

4.3 Abstract Transition Relations

In the previous section we have defined $\widehat{\mathcal{J}} = \langle \widehat{\Sigma}, \widehat{\Sigma}_0, \widehat{\Theta} \rangle$ as the overapproximation of the FTDA instances in $\{\mathcal{I}(n, t, f) \mid f \leq t < n\}$ induced by the abstraction mapping $\delta = \widehat{\alpha}_n \circ \widetilde{\alpha}_n \circ \alpha$, without giving a constructive definition of the abstract transition relation. In this section we show how to efficiently compute abstract versions of the transition relations from Section 3.2: The abstract transitions $\xrightarrow{\widehat{\mathsf{ENV}}}$ and $\xrightarrow{\widehat{\mathsf{MEP}}}$ are straight-forward abstract encodings of updating the environment variables (e.g., crashing processes), and the message exchange phase, respectively. Encoding the abstract process state update $\xrightarrow{\widehat{\mathsf{PROC}}}$ is more involved: due to the counter abstraction, from an index $u \in U$ we have to decode the location that corresponds to that index, and compute the possible successor locations which we store in a relation $\widehat{Active} \subseteq Loc \times Loc$. We use this relation for updating the array $\widehat{\mathbf{active}}$ and the neighborhood arrays $\widehat{\mathbf{Y}} \in \mathit{nbhd}(\widehat{\mathcal{V}})$.

We adapt several notions that we used throughout this paper. First, given an abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ and an index $u \in U$, we say that u *witnesses a process* in $\widehat{\sigma}$ if $u \in [m]$ or if $u \in Loc$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$. Next, we adapt the notions *control*, *row* and *local*. For an abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ and $u \in U$, we denote by:

- $\widehat{\sigma}.control_u$ the tuple $\langle \widehat{\sigma}.\widehat{\mathbf{x}}_1[u], \ldots, \widehat{\sigma}.\widehat{\mathbf{x}}_{|cntl(\widehat{\mathcal{V}})|}[u] \rangle$ if $u \in [m]$, and $u.control$ if $u \in Loc$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$;
- $\widehat{\sigma}.row_u^{\widehat{\mathbf{Y}}}$ the tuple $\langle \widehat{\sigma}.\widehat{\mathbf{Y}}[u, v_1], \ldots, \widehat{\sigma}.\widehat{\mathbf{Y}}[u, v_{|U|}] \rangle \in (2^{D_y})^{|U|}$;
- $\widehat{\sigma}.local_u$ the tuple $\langle \widehat{\sigma}.control_u, \widehat{\sigma}.row_u^{\widehat{\mathbf{Y}}_1}, \ldots, \widehat{\sigma}.row_u^{\widehat{\mathbf{Y}}_{|nbhd(\widehat{\mathcal{V}})|}} \rangle \in \widehat{P}$, where $\widehat{P} = C \times \prod_{\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}})} (2^{D_y})^{|U|}$ is the set of abstract process states;

Abstract environment update. The transition $\widehat{\sigma} \xrightarrow{\widehat{\mathsf{ENV}}} \widehat{\sigma}'$ is defined as follows. First, the predicates from the set Pred are assigned values non-deterministically.

Second, to define the new crashes, for $u \in U$, the value of $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is set to $\{\bot\}$ if u witnesses a failed process, that is, if $u \in [m]$ and $\widehat{\sigma}.\widehat{\mathbf{fld}}[u] = \top$, or if $u \in Loc$ and $u.fld = \top$. Otherwise, if u witnesses a non-failed process, $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is assigned either $\{\bot\}$ or $\{\top\}$ if $u \in [m]$, and one of the values $\{\bot\}$, $\{\top\}$ or $\{\bot, \top\}$ if $u \in Loc$, non-deterministically. If u does not witness a process, $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u] = \emptyset$.

To build the new receiver lists, for every $u, v \in V$ that witness a process, the value of $\widehat{\sigma}'.\widehat{\mathbf{rcv}}[u, v]$ is set to $\{\bot\}$, if v witnesses a failed process. If v witnesses a crashed process, that is, if $\top \in \widehat{\sigma}'.\widehat{\mathbf{cr}}[v]$, then $\widehat{\sigma}'.\widehat{\mathbf{rcv}}[u, v]$ is assigned one of the values $\{\bot\}$, $\{\top\}$ or $\{\bot, \top\}$ non-deterministically. Otherwise, if v witnesses a correct process, $\widehat{\sigma}'.\widehat{\mathbf{rcv}}[u, v] = \{\top\}$. The cells of $\widehat{\sigma}'.\widehat{\mathbf{rcv}}$ indexed by indices from U that do not witness a process are set to $\emptyset$.

Abstract message exchange phase. A transition $\widehat{\sigma} \xrightarrow{\widehat{\mathsf{MEP}}} \widehat{\sigma}'$ is taken if (i) $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v]$ contains $snd_msg(\widehat{\sigma}.control_v)$, for $u, v \in U$ such that $\top \in \widehat{\sigma}.\widehat{\mathbf{rcv}}[u, v]$, (ii) it contains $\bot$, if $\bot \in \widehat{\sigma}.\widehat{\mathbf{rcv}}[u, v]$, and (iii) $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v] = \emptyset$ in the remaining cases.

Abstract process variable update. To define how the control states are updated in the abstract system $\widehat{\mathcal{J}}$, we define abstract guarded assignments. The *abstract guarded assignments* are of the form $\widehat{g} \to \widehat{asg}$, where $\widehat{g}$ is a Boolean combination of *abstract basic guards*, and $\widehat{asg}$ is a partial function, defined in the same way as in the concrete case. We have the following *abstract basic guards*:

guard	notation		evaluation
empty	$g^{\mathbf{true}}$		**true**
control	$g^{x,v}$	where $x \in cntl(V)$ and $v \in D_x$	$control_u.x = v$
termination	$g^{\widehat{pr}}$	where $\widehat{pr}$ abstracts $r \rhd \phi(n, t)$	$\widehat{pr}$
neighborhood	g^{Ξ}	where Ξ is a set of triples $(\widehat{\mathbf{Y}}, \Box, v_y)$ s.t. $\widehat{\mathbf{Y}} \in nbhd(V), \Box \in \{\in, \notin\}$, and $v_y \in D_y$	$\exists v \in U$ $\bigwedge_{\Xi}(v_y \,\Box\, \widehat{\mathbf{Y}}[u, v])$

The abstract guards are evaluated over $local_u$, for $u \in U$. We write $local_u \models \widehat{g}$ if the abstract guard $\widehat{g}$ is satisfied in the abstract process state $local_u$.

The control state update of the fixed m processes is analogous to the concrete case: a set $\widehat{G}_m$ of abstract guarded assignments with pairwise mutually exclusive guards defines a function $update_m : \widehat{P} \to C$.

To update the control states of processes witnessed by $u \in Loc$, we define a set $\widehat{G}_{Loc}$ of guarded assignments, where the guards are not pairwise mutually exclusive. The set $\widehat{G}_{Loc}$ defines a function $update_{Loc}$, which returns a set of control states. Intuitively, processes that are witnessed by the same location may update to different control states in the concrete system, depending on the neighborhood arrays and the environment. This is why, in the set $\widehat{G}_{Loc}$, there can be multiple guarded assignments with the same guard, but different assignments.

Definition 11 **(**$update_{Loc}$**).** *The function $update_{Loc} : \widehat{P} \to 2^C$ maps abstract local states to subsets of the set C of control states. For $u \in Loc$, we define $update_{Loc}(local_u) = \{c \mid \exists \widehat{g} \to \widehat{asg} \in \widehat{G}_{Loc}$ s.t. $local_u \models \widehat{g}$ and $\widehat{asg}$ results in $c\}$.*

To update the array $\widehat{\mathbf{active}}$, we define the following relation.

Definition 12 ($\widehat{Active}$). *A pair (u, u') of locations from Loc are in relation $\widehat{Active} \subseteq Loc \times Loc$ if $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$ and either: (i) $u.failed = \top$ and $u = u'$, or (ii) $u.failed = \bot$ and $\top \in \widehat{\sigma}.\widehat{\mathbf{cr}}[u]$ and $u' = \langle \widehat{\sigma}.control_u, \top \rangle$, or (iii) $u.failed = \bot$ and $\bot \in \widehat{\sigma}.\widehat{\mathbf{cr}}[u]$ and $u' \in update_{Loc}(\widehat{\sigma}.local_u)$.*

The relation $\widehat{Active}$ is used to update the neighborhood arrays $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}}) \setminus \{\widehat{\mathbf{Msg}}\}$, as the update of the locations implies update in the indices of the neighborhood arrays. When updating $\widehat{\mathbf{Y}}$, different cases based on whether u, v are in $[m]$ of in Loc are distinguished. For example, if $u \in [m]$ and $v \in Loc$ then $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v]$ is the union of the sets $\{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u, v_o]\}$ where h_y is the process function for updating $\mathbf{Y}$, $v_o \in Loc$ is the old location that updated to the new location v, and $(v_o, v) \in \widehat{Active}$.

Finally, for two states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{\Sigma}$, it holds that $\widehat{\sigma} \xrightarrow{\widehat{\mathsf{PROC}}} \widehat{\sigma}'$ if:

1. for $u \in [m]$, we have $\widehat{\sigma}'.\widehat{\mathbf{fld}}[u] = \widehat{\sigma}.\widehat{\mathbf{fld}}[u] \vee (\widehat{\sigma}.\widehat{\mathbf{cr}}[u] = \{\top\})$;
2. for $u \in [m]$, we have $\widehat{\sigma}'.control_u = update_m(\widehat{\sigma}.local_u)$ if $\widehat{\sigma}'.\widehat{\mathbf{fld}}[u] = \bot$, and $\widehat{\sigma}'.control_u = \widehat{\sigma}.control_u$ otherwise;
3. for $u \in Loc$, we have $\widehat{\sigma}'.\widehat{\mathbf{active}}[u] = 0$ if $\forall v \in Loc\ (v, u) \notin \widehat{Active}$, and $\widehat{\sigma}'.\widehat{\mathbf{active}}[u] = \mathsf{many}$ otherwise;
4. for $u, v \in U$ and $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}}) \setminus \{\widehat{\mathbf{Msg}}\}$, we have:
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \{h_y(d) \mid d \in \widehat{\mathbf{Msg}}[u, v]\}$, if $u, v \in [m]$;
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \bigcup_{(v_o, v) \in \widehat{Active}} \{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u, v_o]\}$, if $u \in [m], v \in Loc$;
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \bigcup_{(u_o, u) \in \widehat{Active}} \{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u_o, v]\}$, if $u \in Loc, v \in [m]$;
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \bigcup_{\substack{(u_o, u) \in \widehat{Active} \\ (v_o, v) \in \widehat{Active}}} \{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u_o, v_o]\}$, if $u, v \in Loc$.
5. for $u, v \in U$, we have $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v] = \{\bot\}$, if u, v witness a process in $\widehat{\sigma}'$, and $\widehat{\sigma}'.\widehat{\mathbf{Msg}} = \emptyset$ otherwise.

Theorem 2 (Simulation). *Let $\widehat{\mathcal{J}}$ be the overapproximation of $\mathcal{I}(n, t, f)$ induced by the abstraction mapping $\delta = \widehat{\alpha}_n \circ \widetilde{\alpha}_n \circ \alpha$. Suppose $(s, s''') \in \mathcal{Q}(n, t, f)$, such that there exist $s', s'' \in \mathcal{S}(n, t, f)$ with $s \xrightarrow{\mathsf{ENV}} s' \xrightarrow{\mathsf{MEP}} s'' \xrightarrow{\mathsf{PROC}} s'''$. Then it holds that $\delta(s) \xrightarrow{\widehat{\mathsf{ENV}}} \delta(s') \xrightarrow{\widehat{\mathsf{MEP}}} \delta(s'') \xrightarrow{\widehat{\mathsf{PROC}}} \delta(s''')$.*

5 Benchmarks and Experiments

We encoded several synchronous FTDAs from the literature in TLA+ [33] and used the model checker TLC [50]. The experiments were run on a machine with two 12-core Intel(R) Xeon(R) E5-2650 v4 CPUs and 256 GB RAM.

Our benchmarks contain algorithms that solve different variants of the consensus problem, the k-set agreement problem, and the atomic commitment problem;

Table 1: Experimental results for parameterized model checking

algorithm	problem	reference	m	$\mathcal{I}(n,t,f)$ with $t \leq n-m$		m'	$\mathcal{I}(n,t,f)$ with $n-m<t<n$	
				states	time		states	time
FloodSet	consensus	[37, p.103]	2	210 583	2min 28s	1	17 911	11s
FC	fair consensus	[47, p. 17]	2	160 523	3min	1	26 967	18s
EDAC	early deciding consensus	[11]	2	416 120	4h 35min	1	35 027	2min 28s
ESC	early stopping consensus	[47, p. 38]	2	163 772	44min 30s	1	12 784	1min 19s
NBAC	non-blocking atomic commit	[47, p. 82]	2	69 845	40s	1	4 981	5s
FloodMin	k-set agreement, for $k=2$	[37, p.163]	3	10 116 820	10d 16h	2	512 861	1h 39min
						1	43 601	2min 2s

Table 2: Experimental results for the concrete instances of our benchmarks

algorithm	fixed size instance obtained by assigning values to n, t, and f									
	$\mathcal{I}(3,2,1)$		$\mathcal{I}(4,3,2)$		$\mathcal{I}(4,3,3)$		$\mathcal{I}(5,4,2)$		$\mathcal{I}(5,4,3)$	
	states	time	states	time	states	time	states	time	states	time
FloodSet	6 937	4s	99 783	10s	1 220 227	1min 18s	1 024 866	1min 7s	34 724 276	1h 18min
FC	9 118	4s	138 160	11s	1 685 892	1min 45s	1 591 687	1min 39s	53 816 397	1h 43min
EDAC	26 962	5s	242 605	16s	5 703 025	5min 44s	1 940 929	2min 29s	124 183 639	4h 1min
ESC	10 543	4s	170 088	12s	2 954 288	2min 16s	1 577 742	1min 34s	71 913 792	1h 57min
NBAC	256	1s	16 120	7s	16 120	7s	286 670	46s	3 335 753	10min 33s
FloodMin	13 215	6s	287 001	1min 1s	3 311 397	14min 10s	5 297 856	23min 41s	out of memory in 3d 11h	

see the references given in Table 1 for details. As we focus on synchronous algorithms, we have a different set of benchmarks compared to the work in [17,38] that focuses on the partially synchronous algorithms from [12]. The only exception is that [17] considers *FloodMin* in the specific consensus setting ($k = 1$) which boils down to our *FC* consensus benchmark. They check 5 user-provided verification conditions, such as invariants or ranking functions, in less than a second. In our model-checking approach, the user does not have to provide an invariant, thus we have a higher degree of automation.

Table 1 summarizes the experiments for parameterized model checking. In our experiments we assume that the fixed m processes are correct, which implies $f \leq t \leq n - m$. To capture the corner cases $n - m < t < n$ required by the resilience condition $f \leq t < n$, we also do experiments with $m' < m$ concrete processes. In Table 1 we distinguish the cases when at most $m' < m$ are correct (right), from the one where this is not the case (left). We see that most of the verification time is spent on the case of at least m correct processes.

For comparison, Table 2 summarizes the experiments for small instances of up to $n = 5$ processes, where t is set to $n - 1$. We observe that parameterized verification outperforms model checking of fixed size systems already in the case of $n = 5$, $t = 4$, and $f = 3$. In the case of $n = 5$, $t = 4$, and $f = 4$, we were only able to verify the simplest benchmark, *NBAC*. For the remaining ones we reached the limitations of the model checker, as TLC was not able to enumerate all possible successor states due to the immense branching.

By far, *FloodMin* is the most challenging benchmark: its specifications are more complicated, and we therefore have to fix 3 processes (in contrast to 2 in the other benchmarks). In the concrete instance $\mathcal{I}(5, 4, 3)$, i.e., where $n = 5$,

$t = 4$, and $f = 3$, the model checker terminated after three days with an out of memory error.

6 Discussion

While synchronous distributed algorithms are considered "simpler" to design than asynchronous ones, encoding and model checking synchronous algorithms is a challenge: All processes take steps simultaneously, and each process can transfer into several successor states depending on the received messages, which are subject to non-determinism by the environment. We noticed in our experiments that synchronously selecting a successor state for each process combined with the non-determinism results in a huge branching factor. In conjunction with the additional non-determinism introduced through abstraction, this poses serious challenges to the explicit state model checker TLC. In future work, we will consider other model checking back-ends, and different encodings. Our predicate abstraction currently requires some domain knowledge to capture the interplay of the number of faults and round numbers. As future work we consider automatic generation of this abstraction by means of static analysis on the environment. All other abstractions can be done automatically. Finally, more complex resilience conditions that appear in the literature, such as $n > 2t$, would require a finer abstraction than the one we present here, a topic that we reserve for future work.

Parameterized model checking is undecidable in general [3,4,6,19,49]. Still, there are techniques for specific classes of systems. A popular technique is abstraction. Different domain-specific abstractions have been used for mutual exclusion [15,16,46], cache coherence [13,32,40,43], dynamic scheduling [39], and recently to asynchronous FTDAs [2,27,28,29]. Most of these parameterized model checking techniques consider asynchronous systems. The work most closely related to ours are the cutoff results of [38], as (i) it targets at completely automated verification, and (ii) while we have simulation to abstract systems, the authors of [38] prove simulation to small systems. To achieve this, the authors had to restrict the fragment to which the cutoff theorem applies: First, the cutoff only applies to consensus algorithms, that is, to three specific LTL specifications. As noted in [38], generalizing this to other specifications, e.g., k-set agreement, non-blocking atomic commit, or even a more complete logic fragment would require more theoretical work. Our case studies discussed in Section 5 include other algorithms than just consensus. Second, the guarded command language introduced in [38] can express only threshold guards containing predicates on the number of messages received by a process in the current round. However, there are several round-based distributed algorithms, in particular synchronous ones, that contain other guards; for instance, *termination guards* that check whether a given round number is reached, or guards that check whether messages from the same set of processes are received in two consecutive rounds. Our guarded commands contain such guards. Still, we currently cannot express all distributed algorithms, and extending our verification methods to other syntactic constructs is future work.

References

[1] C. Aiswarya, Benedikt Bollig, and Paul Gastin. An automata-theoretic approach to the verification of distributed algorithms. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 340–353, 2015.

[2] Francesco Alberti, Silvio Ghilardi, Andrea Orsini, and Elena Pagani. Counter Abstractions in Model Checking of Distributed Broadcast Algorithms: Some Case Studies. In *Proceedings of the 31st Italian Conference on Computational Logic, Milano, Italy, June 20-22, 2016.*, pages 102–117, 2016.

[3] Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. Parameterized Model Checking of Rendezvous Systems. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 109–124, 2014.

[4] Krzysztof R. Apt and Dexter Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[5] Hagit Attiya and Jennifer Welch. *Distributed Computing.* Wiley, 2nd edition, 2004.

[6] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[7] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Reasoning about Networks with Many Identical Finite State Processes. *Inf. Comput.*, 81(1):13–31, 1989.

[8] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350, 2006.

[9] Armando Castañeda, Yoram Moses, Michel Raynal, and Matthieu Roy. Early Decision and Stopping in Synchronous Consensus: A Predicate-Based Guided Tour. In *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, pages 206–221, 2017.

[10] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *Reachability Problems, 3rd International Workshop, RP 2009, Palaiseau, France, September 23-25, 2009. Proceedings*, pages 93–106, 2009.

[11] Bernadette Charron-Bost and André Schiper. Uniform Consensus is Harder than Consensus. *J. Algorithms*, 51(1):15–37, 2004.

[12] Bernadette Charron-Bost and André Schiper. The Heard-Of Model: Computing in Distributed Systems with Benign Faults. *Distributed Computing*, 22(1):49–71, 2009.

[13] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pages 382–398. 2004.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[15] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Environment Abstraction for Parameterized Verification. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 126–141, 2006.

[16] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. 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*, pages 33–47, 2008.

[17] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A Logic-Based Framework for Verifying Consensus Algorithms. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, pages 161–181, 2014.

[18] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 400–415, 2016.

[19] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about Rings. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 85–94, 1995.

[20] E. Allen Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[21] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[22] Dana Fisman, Orna Kupferman, and Yoad Lustig. On Verifying Fault Tolerance of Distributed Protocols. 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*, pages 315–331, 2008.

[23] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

[24] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 201–209, 2013.

[25] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms. In *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*, pages 209–226, 2013.

[26] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 179–188, 2007.

[27] Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. Para^2: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. *Formal Methods in System Design*, 51(2):270–307, 2017.

[28] Igor V. Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant

Distributed Algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 719–734, 2017.

[29] Igor V. Konnov, Helmut Veith, and Josef Widder. On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability. *Inf. Comput.*, 252:95–109, 2017.

[30] Hermann Kopetz and Günter Grünsteidl. TTP - A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, 1994.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.

[32] Sava Krstić. Parametrized System Verification with Guard Strengthening and Parameter Abstraction. In *AVIS*, 2005.

[33] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[34] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[35] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 357–370, 2016.

[36] Patrick Lincoln and John M. Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Digest of Papers: FTCS-23, The Twenty-Third Annual International Symposium on Fault-Tolerant Computing, Toulouse, France, June 22-24, 1993*, pages 402–411, 1993.

[37] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[38] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff Bounds for Consensus Algorithms. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 217–237, 2017.

[39] Kenneth L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 219–234, 1999.

[40] Kenneth L. McMillan. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, pages 179–195, 2001.

[41] Andre Medeiros. ZooKeeper's atomic broadcast protocol: Theory and practice. Technical report, 2012.

[42] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372, 2013.

[43] John W. O'Leary, Murali Talupur, and Mark R. Tuttle. Protocol Verification Using Flows: An Industrial Experience. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 172–179, 2009.

[44] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.

[45] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making Fast Consensus Generally Faster. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 156–167, 2016.
[46] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 107–122, 2002.
[47] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.
[48] Wilfried Steiner, John M. Rushby, Maria Sorea, and Holger Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 189–198, 2004.
[49] Ichiro Suzuki. Proving Properties of a Ring of Finite-State Machines. *Inf. Process. Lett.*, 28(4):213–214, 1988.
[50] TLA+ Toolbox. `http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html`.
[51] Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.
[52] Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and Universal Quantifiers for Verifying Parameterized Systems. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 599–613, 2016.
[53] Josef Widder, Günther Gridling, Bettina Weiss, and Jean-Paul Blanquart. Synchronous Consensus with Mortal Byzantines. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 102–112, 2007.
[54] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for Change in a Formal Verification of the RAFT Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 154–165, 2016.
[55] Apache ZooKeeper. Web page. `http://zookeeper.apache.org/`.

Gradual Program Verification

Johannes Bader[1], Jonathan Aldrich[2], and Éric Tanter[3]

[1] Microsoft Corporation, Redmond, USA,
jobader@microsoft.com
[2] Institute for Software Research, Carnegie Mellon University, Pittsburgh, USA
jonathan.aldrich@cs.cmu.edu
[3] PLEIAD Lab, Computer Science Dept (DCC), University of Chile, Santiago, Chile
etanter@dcc.uchile.cl

Abstract. Both static and dynamic program verification approaches have significant disadvantages when considered in isolation. Inspired by research on gradual typing, we propose *gradual verification* to seamlessly and flexibly combine static and dynamic verification. Drawing on general principles from abstract interpretation, and in particular on the recent Abstracting Gradual Typing methodology of Garcia *et al.*, we systematically derive a gradual verification system from a static one. This approach yields, by construction, a gradual verification system that is compatible with the original static system, but overcomes its rigidity by resorting to dynamic verification when desired. As with gradual typing, the programmer can control the trade-off between static and dynamic checking by tuning the (im)precision of pre- and postconditions. The formal semantics of the gradual verification system and the proofs of its properties, including the gradual guarantees of Siek *et al.*, have been fully mechanized in the Coq proof assistant.

1 Introduction

Program verification techniques have the potential to improve the correctness of programs, by exploiting pre- and postconditions specified in formulas drawn from a given logic, such as Hoare logic [8]. Unfortunately, traditional approaches to verification have a number of shortcomings, as illustrated next.

Example 1.

```
int withdraw(int balance, int amount)
    requires (balance ≥ amount) ensures (balance ≥ 0) {
    return balance - amount; // returns the new balance
}

int balance := 100;
balance := withdraw(balance, 30);
balance := withdraw(balance, 40);
```

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 25–46, 2018.
https://doi.org/10.1007/978-3-319-73721-8_2

In this case, we reason about a variable `balance` representing some bank account. The contract (pre- and postconditions) of `withdraw` specifies that it may only be called if the balance is high enough to withdraw the given amount, ensuring that no negative balance is reached. There are a number of ways to verify Example 1. We briefly discuss static and dynamic verification, including hybrid approaches. We then introduce *gradual verification* as an approach that has the potential to overcome a number of their shortcomings.

Static verification. Formal methods like Hoare logic are used to establish *statically* that a program is *valid*, *i.e.* satisfies its specification. In Example 1, the static verifier proves both that `withdraw` itself complies with its contract and that the three statements below are valid, *e.g.* that the precondition of `withdraw` is satisfied prior to both calls.

A lack of detailed contracts may prevent the verifier from establishing that a program is valid. In Example 1, verification of the second call to `withdraw` in fact fails: after the first call, the verifier knows from the postcondition that `(balance ≥ 0)`, which is insufficient to justify that `(balance ≥ 40)` as required for the second call. Deriving such knowledge would require a stronger postcondition such as `balance = old(balance) - amount`. However, this is not the postcondition that was provided by the programmer, perhaps intentionally (*e.g.* if the intent was to focus on some weaker correctness properties) or perhaps due to limited expressiveness of the underlying logic (notation such as `old(`x`)` may not exist). In general, a verification tool might also fail to prove program properties due to undecidability of the underlying logic or practical limitations of the specific tool implementation.

Hoare logic has been extended to more powerful logics like separation logic [15] and implicit dynamic frames [20]. Yet, the requirement of rigorous annotation of contracts remains an issue in these settings. Due to space limitations and to capture the core ideas of gradual verification, this paper focuses on a simple Hoare logic. We have formalized an extension to implicit dynamic frames and implemented a prototype, which can both be found at http://olydis.github.io/GradVer/impl/HTML5wp/

Dynamic verification. An alternative approach is to use *dynamic* verification to ensure that a program adheres to its specification at runtime, by turning the contract into *runtime checks*. A contract violation causes a runtime exception to be thrown, effectively preventing the program from entering a state that contradicts its specification. In Example 1, a dynamic verification approach would not raise any error because the `balance` is in fact sufficient for both calls to succeed. Note that because contracts are checked at runtime, one can even use arbitrary programs as contracts, and not just formulas drawn from a logic [6].

Meyer's Design by Contract methodology [12] integrated writing contracts in this way as an integral part of the design process, with the Eiffel language automatically performing dynamic contract verification [11]. Dynamic verification has also notably been used to check JML specifications [3], and has been extended to the case of separation logic by Nguyen *et al.* [14]. Note that unlike the static approach, the dynamic approach only requires programmers to encode the

properties they care about as pre- and postconditions, and does not require extra work for the sake of avoiding false negatives. However, the additional checks impose runtime overhead that may not always be acceptable. Furthermore, violations of the specification are no longer detected ahead of time.

Hybrid approaches. Recognizing that static and dynamic checking have complementary advantages, some approaches to combine them have emerged. In particular, with the Java Modeling Language (JML) [2] and Code Contracts [5], it is possible to use the same specifications for either static or dynamic verification. Additionally, Nguyen *et al.* explored a hybrid approach to reduce the overhead of their approach to runtime checking for separation logic, by exploiting static information [14].

Although useful, these techniques do not support a smooth continuum between static and dynamic verification. With the JML approach, engineers enable static *or* dynamic verification; the two checking regimes do not interact. Nguyen *et al.* use the static checker to optimize runtime checks, but do not try to report static verification failures because it is difficult to distinguish failures due to contradictions in the specification (which the developer should be warned about) from failures due to leaving out parts of the specification (which could have been intentional underspecification, and thus should not produce a warning). Their runtime checking approach also requires the specification of heap footprints to match in pre- and post-conditions, which like many static checking approaches forces programmers to do extra specification work to avoid false negatives.

Gradual verification. Because this tension between static and dynamic verification is reminiscent of the tension between static and dynamic type checking, we propose to draw on research on *gradual typing* [18,17,7] to develop a flexible approach to program verification, called *gradual verification*. Gradual typing supports both static and dynamic checking and the entire spectrum in between, driven by the precision of programmer annotations [19]. Similarly, gradual verification introduces a notion of *imprecise contracts*, supporting a continuum between static and dynamic verification. A static checker can analyze a gradually-specified program and warn the programmer of inconsistencies between specifications and code, including contracts that are intended to be fully precise but are not strong enough, as well as contracts that contradict one another despite possible imprecision in each. On the other hand, the static checker will not produce warnings that arise from a contract that is intentionally imprecise; in these cases, runtime checking is used instead. Programmers can rely on a *gradual guarantee* stating that reducing the precision of specifications never breaks the verifiability (and reduceability) of a program. This guarantee, originally formulated by Siek *et al.* in the context of gradual types [19], ensures that programmers can choose their desired level of precision without artificial constraints imposed by the verification technology.

It is worth noting that the similarly named work “The Gradual Verifier” [1] focuses on *measuring the progress of static verification*. Their verification technique “GraVy” is neither sound nor complete and does not comply with the gradual guarantee.

Gradual verification is not only useful in cases of missing information (*e.g.* when reusing a library that is not annotated) but also to overcome limitations of the static verification system as motivated by Example 1. Furthermore, programmers can gradually evolve and refine static annotations. As they do so, they are rewarded by progressively *increased static correctness guarantees* and progressively *decreased runtime checking*, supporting a pay-as-you-go cost model.

Specifically, we support imprecision by introducing an unknown formula `?` that acts like a wildcard during static verification. Semantically, the static checker will optimistically accept a formula containing `?` as long as there exists some interpretation of `?` that makes the formula valid. As we learn more information about the program state at runtime, the dynamic checker ensures that some valid instantiation still exists. Crucially, the unknown formula can be combined with static formulas, forming *imprecise* formulas. For instance, going back to Example 1, we can write the imprecise postcondition $(\texttt{balance} \geq 0) \wedge ?$ in order to enable gradual reasoning, resulting in an optimistic interpretation of `?` as $(\texttt{balance} \geq 40)$ when statically proving the precondition of the second call. At runtime, this interpretation is checked, to ensure soundness.

Note that the postcondition we suggest is only *partially unknown*, preserving the *static knowledge* $(\texttt{balance} \geq 0)$. This not only allows us to prove certain goals (*e.g.* $(\texttt{balance} \neq -10)$) without requiring any dynamic checks, but also to statically reject programs that provably contradict this knowledge (*e.g.* if a subsequent call had `balance = -10` as precondition).

Contributions. This paper is the first development of the ideas of gradual typing in the context of program logics for verification. More precisely, we first introduce a simple statically-verified language called SVL, along with its associated program logic. We then adapt the *Abstracting Gradual Typing* methodology (AGT) [7] to the verification setting and show in section 3 how the static semantics of a gradually-verified language GVL can be derived from SVL using principles of abstract interpretation. Section 4 develops GVL's dynamic semantics. Here, we deviate from the AGT approach and instead propose injecting a minimal amount of runtime assertion checks, yielding a pay-as-you-go cost model. Finally, Section 5 briefly discusses GVL_{IDF}, an extension of our approach to heap-allocated objects and an extended logic with implicit dynamic frames [20].

Limitations. Our approach for dynamic semantics requires assertions to be evaluable at runtime, naturally limiting the logic usable for annotations. The AGT methodology (based on combining the proof-trees at runtime) is not restricted that way, so it may be the ideal starting point for gradual verification in presence of higher-order logic assertions.

The formal semantics of GVL and the proofs of its properties have been fully mechanized in the Coq proof assistant and can be found at `http://olydis.github.io/GradVer/impl/HTML5wp/`. The site also includes a report with the formal treatment of the extended logic, as well as an interactive online prototype of GVL_{IDF}. Due to limited space, some figures contain only selected parts of definitions. Complete definitions can be found online as well.

$$\begin{aligned}
\textit{program} &::= \overline{\textit{procedure}}\ s \\
\textit{procedure} &::= T\ m(\overline{T\ x})\ \textit{contract}\ \texttt{\{}\ s\ \texttt{\}} \\
\textit{contract} &::= \texttt{requires}\ \phi\ \texttt{ensures}\ \phi \\
T &::= \texttt{int} \\
\oplus &::= \texttt{+} \mid \texttt{-} \mid \ldots \\
\odot &::= \texttt{=} \mid \neq \mid \texttt{<} \mid \ldots
\end{aligned}$$

$$\begin{aligned}
s \in \textsc{Stmt} &::= \texttt{skip} \mid s_1\texttt{;}\ s_2 \mid T\ x \mid x\ \texttt{:=}\ e \\
&\quad\mid x\ \texttt{:=}\ m(x) \mid \texttt{assert}\ \phi \\
e \in \textsc{Expr} &::= v \mid x \mid (e\ \oplus\ e) \\
x \in \textsc{Var} &::= \texttt{result} \mid \textit{ident} \mid \texttt{old}(\textit{ident}) \\
v \in \textsc{Val} &::= n \\
\phi \in \textsc{Formula} &::= \texttt{true} \mid (e\ \odot\ e) \mid \phi \land \phi
\end{aligned}$$

and syntactic sugar $\texttt{return}\ e \stackrel{\text{def}}{=} \texttt{result :=}\ e$ and $T\ x\ \texttt{:=}\ e \stackrel{\text{def}}{=} T\ x\texttt{;}\ x\ \texttt{:=}\ e$

Fig. 1. SVL: Syntax

2 SVL: Statically Verified Language

In the following sections, we describe a simple statically verified language called SVL. We formalize its syntax, semantics and soundness.

2.1 Syntax

Figure 1 shows the syntax of SVL. Programs consist of a collection of procedures and a designated statement resembling the entry point ("main procedure"). We include the empty statement, statement sequences, variable declarations, variable assignments, procedure calls, and assertions. All statements are in A-normal form, which is not essential semantically but does simplify the formalism. Procedures have contracts consisting of a pre- and postcondition, which are formulas. Formulas can be the constant `true`, binary relations between expressions, and a conjunction $\land$. Expressions can occur within a formula or variable assignment, and consist of variables, constants and arithmetic operations. [4]

For the remainder of this work we only consider well-formed programs: variables are declared before use, procedure names are unique and contracts only contain variables that are in scope. More specifically, a precondition may only contain the procedure's parameters x_i, a postcondition may only contain the special variable `result` and the dummy variables $\texttt{old}(x_i)$.

To simplify the presentation of semantics, we will give rules for procedures that have exactly one parameter.

2.2 Dynamic Semantics

We now describe the dynamic semantics of SVL. SVL has a small-step semantics $\cdot \longrightarrow \cdot : \textsc{State} \rightharpoonup \textsc{State}$ (see Fig. 2) that describes discrete transitions between program states. Program states that are not in the domain of this partial function are said to be *stuck*, which happens if an assertion does not hold or a contract is

[4] Our approach is directly applicable to, say, further control structures, a richer type system or formulas that are arbitrary boolean expressions.

violated before/after a call. In Section 2.3, we define a static verification system whose soundness result implies that valid SVL programs do not get stuck.

Program states. Program states consist of a stack, *i.e.* $\textsc{State} = \textsc{Stack}$ where:

$$S \in \textsc{Stack} ::= E \cdot S \mid \mathsf{nil} \qquad \text{where} \qquad E \in \textsc{StackFrame} = \textsc{Env} \times \textsc{Stmt}$$

A stack frame consists of a local variable environment $\rho \in \textsc{Env} = \textsc{Var} \rightharpoonup \textsc{Val}$ and a continuation statement.

Evaluation. An expression e is evaluated according to a big-step evaluation relation $\rho \vdash e \Downarrow v$, yielding value v using local variable environment $\rho \in \textsc{Env}$ of the top-most stack-frame. The definition is standard: variables are looked up in ρ, and the resulting values are combined according to standard arithmetic rules. Example: $[\mathtt{x} \mapsto 3] \vdash \mathtt{x\ +\ 5} \Downarrow 8$

The evaluation of a formula in a given environment is specified by the predicate $\cdot \models \cdot \subseteq \textsc{Env} \times \textsc{Formula}$. We assume standard evaluation semantics for standard concepts like equality. We also say that a formula *describes* a certain (infinite) set of environments (exactly the environments under which it holds), yielding natural definitions for formula satisfiability and implication.

Definition 1 (Denotational Formula Semantics).
Let $[\![\cdot]\!] : \textsc{Formula} \rightarrow \mathcal{P}(\textsc{Env})$ *be defined as* $[\![\phi]\!] \stackrel{\mathsf{def}}{=} \{\ \rho \in \textsc{Env} \mid \rho \models \phi\ \}$

Definition 2 (Formula Satisfiability). *A formula* ϕ *is satisfiable if and only if* $[\![\phi]\!] \neq \emptyset$*. Let* $\textsc{SatFormula} \subset \textsc{Formula}$ *be the set of satisfiable formulas.*

Definition 3 (Formula Implication). $\phi_1 \Rightarrow \phi_2$ *if and only if* $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$

Reduction rules. We define a standard small-step reduction semantics for statements (Fig. 2). SsAssert ensures that assertions are stuck if the asserted formula is not satisfied. SsCall sets up a new stack frame and makes sure that the procedure's precondition is satisfied by the newly set up context. Similarly, SsCallExit ensures that the postcondition is satisfied before returning control to the call site. Note our use of auxiliary functions procedure and mpost in order to retrieve a procedure's definition or postcondition using that procedure's name. Formally, we assume all rules and definitions are implicitly parameterized with the "executing" program $p \in \textsc{Program}$ from which to extract this information. When required for disambiguation, we explicitly annotate reduction arrows with the executing program p, as in $\longrightarrow_p$.

Note that SsCall also initializes $\mathtt{old}(x')$, which allows assertions and most importantly the postcondition to reference the parameter's initial value. In reality, no additional memory is required to maintain this value since it is readily available as $\rho(x)$, i.e. it lives in the stack frame of the call site. For a program to be well-formed, it may not write to $\mathtt{old}(x')$ in order to enable this reasoning.

$$\frac{\rho \vDash \phi}{\langle \rho, \texttt{assert}\ \ \phi \rangle \cdot S \longrightarrow \langle \rho, \texttt{skip} \rangle \cdot S}\ \textsc{SsAssert}$$

$$\frac{\rho \vdash e \Downarrow v \qquad \rho' = \rho[x \mapsto v]}{\langle \rho, x\ \texttt{:=}\ e \rangle \cdot S \longrightarrow \langle \rho', \texttt{skip} \rangle \cdot S}\ \textsc{SsAssign}$$

$$\frac{\begin{array}{c}\mathsf{procedure}(m) = T_r\ m\texttt{(}T\ x'\texttt{)}\ \texttt{requires}\ \phi_p\ \texttt{ensures}\ \phi_q\ \texttt{\{}\ r\ \texttt{\}} \\ \rho \vdash x \Downarrow v \qquad \rho' = [x' \mapsto v, \texttt{old(}x'\texttt{)} \mapsto v] \qquad \rho' \vDash \phi_p\end{array}}{\langle \rho, y\ \texttt{:=}\ m\texttt{(}x\texttt{);}\ s \rangle \cdot S \longrightarrow \langle \rho', r \rangle \cdot \langle \rho, y\ \texttt{:=}\ m\texttt{(}x\texttt{);}\ s \rangle \cdot S}\ \textsc{SsCall}$$

$$\frac{\mathsf{post}(m) = \phi_q \qquad \rho' \vDash \phi_q}{\langle \rho', \texttt{skip} \rangle \cdot \langle \rho, y\ \texttt{:=}\ m\texttt{(}x\texttt{);}\ s \rangle \cdot S \longrightarrow \langle \rho[y \mapsto \rho'(\texttt{result})], s \rangle \cdot S}\ \textsc{SsCallExit}$$

Fig. 2. SVL: Small-step semantics (selected rules)

$$\begin{array}{llll}
\mathsf{WLP}(\texttt{skip}, \phi) & = \phi & \mathsf{WLP}(s_1\texttt{;}\ s_2, \phi) & = \mathsf{WLP}(s_1, \mathsf{WLP}(s_2, \phi)) \\
\mathsf{WLP}(x\ \texttt{:=}\ e, \phi) & = \phi[e/x] & \mathsf{WLP}(\texttt{assert}\ \ \phi_a, \phi) & = \phi_a\ \wedge\ \phi
\end{array}$$

$$\begin{aligned}
\mathsf{WLP}(y\ \texttt{:=}\ m\texttt{(}x\texttt{)}, \phi) = \max_{\Rightarrow}\ \{\ \phi' \mid\ & y \notin \mathsf{FV}(\phi') \wedge \\
& \phi' \Rightarrow \mathsf{mpre}(m)[x/\mathsf{mparam}(m)] \wedge \\
& (\phi'\ \wedge\ \mathsf{mpost}(m)[x, y/\texttt{old(}\mathsf{mparam}(m)\texttt{)}, \texttt{result}]) \Rightarrow \phi\ \}
\end{aligned}$$

Fig. 3. SVL: Weakest precondition (selected rules)

2.3 Static Verification

We define the static verification of SVL contracts through a weakest liberal precondition calculus [4]. This syntax-directed approach (compared to, say, Hoare logic, which has an existential in its sequence rule) will be useful for the dynamic semantics of our gradual language (will be pointed out again later).

Definition 4 (Valid Procedure).
A procedure with contract `requires` ϕ_p `ensures` ϕ_q*, parameter* x *and body* s *is considered valid if* $\phi_p \Rightarrow \mathsf{WLP}(s, \phi_q)[x/\texttt{old(}x\texttt{)}]$ *holds.*

We define $\mathsf{WLP} : \textsc{Stmt} \times \textsc{Formula} \rightharpoonup \textsc{Formula}$ as shown in Figure 3. WLP is standard for the most part. The rule for calls computes a maximal formula ϕ' (*i.e.* minimum information content) that is sufficient to imply both the procedure's precondition and the overall postcondition ϕ with the help of the procedure's postcondition.

Definition 5 (Valid Program). *A program with entry point statement* s *is considered valid if* `true` $\Rightarrow \mathsf{WLP}(s, \texttt{true})$ *holds and all procedures are valid.* [5]

[5] Note that one can demand more than `true` to hold at the final state by simply ending the program with an assertion statement.

$$\mathsf{sWLP}(s \cdot \mathsf{nil}, \phi) = \mathsf{WLP}(s, \phi) \cdot \mathsf{nil}$$
$$\mathsf{sWLP}(s \cdot (y \texttt{ := } m(x)\texttt{;}\ s') \cdot \overline{s}, \phi) = \mathsf{WLP}(s, \mathsf{mpost}(m)) \cdot \mathsf{sWLP}((y \texttt{ := } m(x)\texttt{;}\ s') \cdot \overline{s}, \phi)$$

Fig. 4. Weakest precondition across call boundaries

Example 2 (Static Checker of SVL). We demonstrate the resulting behavior of SVL's static checker using example 1, but with varying contracts:

requires (balance ≥ amount)
ensures (result = old(balance) - old(amount))
withdraw is valid since the WLP of the body, given the postcondition, is (balance - amount = old(balance) - old(amount)). Substitution gives (balance - amount = balance - amount) which is trivially implied by the precondition. The overall program is also valid since the main procedure's WLP is (100 ≥ 70) which is implied by true.

requires (balance ≥ amount) ensures (result ≥ 0) (as in example 1)
withdraw is valid since the body's WLP is (balance - amount ≥ 0) which matches the precondition. However, the program is not valid: The WLP of the second call is (balance ≥ 40) which is not implied by the postcondition of the first call. As a result, the WLP of the entire main procedure is undefined.

requires (balance ≥ 0) ensures (result ≥ 0)
Validating withdraw fails since the body's WLP (same as above) is not implied by the precondition.

2.4 Soundness

Verified programs should respect contracts and assertions. We have formulated the runtime semantics of SVL such that they get stuck if contracts or assertions are violated. As a result, *soundness* means that valid programs do not get stuck. In particular, we can use a syntactic progress/preservation argument [22].

If the WLP of a program is satisfied by the current state, then execution will not get stuck (progress) and after each step of execution, the WLP of the remaining program is again satisfied by the new state (preservation). We use a progress and preservation formulation of soundness not just because it allows us to reason about non-terminating programs, but mainly because this will allow us to naturally deal with the runtime checking needs of gradual verification.

To simplify reasoning about states with multiple stack frames, we extend the definition of WLP to accept a stack of statements and return a stack of preconditions, as shown in Figure 4. Note that WLP as defined previously can only reason about procedure calls atomically since an element of STMT cannot encode intermediate states of an ongoing procedure call. In contrast sWLP works across call boundaries by accepting a stack of statements and recursively picking up the postconditions of procedures which are currently being executed.

While before we defined what makes procedures as a whole valid, we can now validate arbitrary intermediate program states, e.g. we can say that

$$\mathsf{sWLP}\left(\begin{array}{c}\texttt{return balance - amount}\\ \cdot \\ \texttt{b2 := withdraw(b1,a)}\\ \cdot \\ \mathsf{nil}\end{array}, \texttt{(b2} \neq \texttt{-1)} \wedge \texttt{(a = 4)}\right) = \begin{array}{c}\texttt{(balance - amount} \geq \texttt{0)}\\ \cdot \\ \texttt{(b1} \geq \texttt{a)} \wedge \texttt{(a = 4)}\\ \cdot \\ \mathsf{nil}\end{array}$$

where `withdraw` is defined as in example 1. If $\overline{s}$ are the continuations of some arbitrary program state $\pi \in \text{STATE}$, then $\mathsf{sWLP}(\overline{s}, \texttt{true})$ is the precondition for $\overline{s}$. If $\mathsf{sWLP}(\overline{s}, \texttt{true})$ holds in the variable environments $\overline{\rho}$ of π, respectively, then soundness guarantees that the program does not get stuck. In the following, we extend the notion of validity to arbitrary intermediate program states in order to formalize progress and preservation. Validity of states is an invariant that relates the static and dynamic semantics of valid SVL programs.

Definition 6 (Valid state). *We call the state* $\langle \rho_n, s_n \rangle \cdot ... \cdot \langle \rho_1, s_1 \rangle \cdot \mathsf{nil} \in \text{STATE}$ valid *if* $\rho_i \vDash \mathsf{sWLP}_i(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \texttt{true})$ *for all* $1 \leq i \leq n$. *(*$\mathsf{sWLP}_i(\overline{s}, \phi)$ *is the* i*-th component of* $\mathsf{sWLP}(\overline{s}, \phi)$*)*

Validity of the initial program state follows from validity of the program (Def. 5).

Proposition 1 (SVL: Progress). *If* $\pi \in \text{STATE}$ *is a valid state and* $\pi \notin \{\langle \rho, \texttt{skip} \rangle \cdot \mathsf{nil} \mid \rho \in \text{ENV}\}$ *then* $\pi \longrightarrow \pi'$ *for some* $\pi' \in \text{STATE}$.

Proposition 2 (SVL: Preservation). *If* π *is a valid state and* $\pi \longrightarrow \pi'$ *for some* $\pi' \in \text{STATE}$ *then* π' *is a valid state.*

3 GVL: Static Semantics

Having defined SVL, we can now derive its gradual counterpart GVL, which supports gradual program verification thanks to *imprecise* contracts. We follow the abstract interpretation perspective on gradual typing [7], AGT for short. In this sense, we introduce *gradual formulas* as formulas that can include the *unknown formula*, denoted `?`:

$\widetilde{\phi} ::= \phi \mid \phi \wedge \texttt{?}$ and standalone formula `?` as syntactic sugar for $\texttt{true} \wedge \texttt{?}$

We define $\widetilde{\text{FORMULA}}$ as the set of all gradual formulas. The syntax of GVL is unchanged save for the use of gradual formulas in contracts: *contract* ::= $\texttt{requires}\ \widetilde{\phi}\ \texttt{ensures}\ \widetilde{\phi}$. In Sections 3.2 to 3.4 we *lift* the predicates and functions SVL uses for verification from the static domain to the gradual domain, yielding a gradual verification logic for GVL.

3.1 Interpretation of Gradual Formulas

We call ϕ in $\phi \wedge \texttt{?}$ *static part* of the imprecise formula and define a helper function $\mathsf{static} : \widetilde{\text{FORMULA}} \rightarrow \text{FORMULA}$ that extracts the static part of a gradual formula, i.e. $\mathsf{static}(\phi) = \phi$ and $\mathsf{static}(\phi \wedge \texttt{?}) = \phi$. Following the AGT approach, a gradual formula is given meaning by concretization to *the set of static formulas* that it represents.

Definition 7 (Concretization of gradual formulas).
$\gamma : \widetilde{\text{FORMULA}} \rightharpoonup \mathcal{P}^{\text{FORMULA}}$ *is defined as:*

$$\gamma(\phi) = \{ \phi \}$$
$$\gamma(\phi \ \wedge \ ?) = \{ \phi' \in \text{SATFORMULA} \mid \phi' \Rightarrow \phi \} \quad \text{if } \phi \in \text{SATFORMULA}$$
$$\gamma(\phi \ \wedge \ ?) \quad \text{undefined otherwise}$$

A fully-precise formula concretizes to the singleton set. Importantly, we only concretize imprecise formulas to precise formulas that are *satisfiable*. Note that the concretization of any gradual formula always implies the static part of that formula. The notion of *precision* between formulas, reminiscent of the notion of precision between gradual types [19], is naturally induced by concretization [7]:

Definition 8 (Precision). *$\widetilde{\phi}_1$ is more precise than $\widetilde{\phi}_2$, written $\widetilde{\phi}_1 \sqsubseteq \widetilde{\phi}_2$ if and only if $\gamma(\widetilde{\phi}_1) \subseteq \gamma(\widetilde{\phi}_2)$.*

3.2 Lifting Predicates

The semantics of SVL makes use of predicates that operate on formulas, namely formula implication and formula evaluation. As GVL must operate on gradual formulas, these predicates are lifted in order to deal with gradual formulas in a consistent way. We propose the following definitions of consistent formula evaluation and implication.

Definition 9 (Consistent Formula Evaluation).
Let $\cdot \,\widetilde{\vDash}\, \cdot \subseteq \text{ENV} \times \widetilde{\text{FORMULA}}$ *be defined as* $\rho \,\widetilde{\vDash}\, \widetilde{\phi} \iff \rho \vDash \mathsf{static}(\widetilde{\phi})$

Definition 10 (Consistent Formula Implication).
Let $\cdot \,\widetilde{\Rightarrow}\, \cdot \subseteq \widetilde{\text{FORMULA}} \times \widetilde{\text{FORMULA}}$ *be defined inductively as*

$$\frac{\phi_1 \Rightarrow \mathsf{static}(\widetilde{\phi}_2)}{\phi_1 \,\widetilde{\Rightarrow}\, \widetilde{\phi}_2}\ \widetilde{\text{IMPLSTATIC}} \qquad \frac{\phi \in \text{SATFORMULA} \quad \phi \Rightarrow \phi_1 \quad \phi \Rightarrow \mathsf{static}(\widetilde{\phi}_2)}{\phi_1 \ \wedge \ ? \,\widetilde{\Rightarrow}\, \widetilde{\phi}_2}\ \widetilde{\text{IMPLGRAD}}$$

In rule $\widetilde{\text{IMPLGRAD}}$, ϕ represents a *plausible* formula represented by $\phi_1 \ \wedge \ ?$.

Abstract interpretation. Garcia *et al.* [7] define consistent liftings of predicates as their *existential* liftings:

Definition 11 (Consistent Predicate Lifting). *The consistent lifting $\widetilde{P} \subseteq \widetilde{\text{FORMULA}} \times \widetilde{\text{FORMULA}}$ of a predicate $P \subseteq \text{FORMULA} \times \text{FORMULA}$ is defined as:*

$$\widetilde{P}(\widetilde{\phi}_1, \widetilde{\phi}_2) \overset{\mathsf{def}}{\iff} \exists \phi_1 \in \gamma(\widetilde{\phi}_1), \phi_2 \in \gamma(\widetilde{\phi}_2).\ P(\phi_1, \phi_2)$$

Our definitions above are proper predicate liftings.

Lemma 1 (Consistent Formula Evaluation and Implication).
$\cdot \,\widetilde{\vDash}\, \cdot$ (Def. 9) is a consistent lifting of $\cdot \vDash \cdot$ and $\cdot \,\widetilde{\Rightarrow}\, \cdot$ (Def. 10) is a consistent lifting of $\cdot \Rightarrow \cdot$.

3.3 Lifting Functions

Deriving gradual semantics from SVL also involves lifting *functions* that operate on formulas, most importantly WLP (Definition 3). Figure 5 gives the definition of $\widetilde{\mathsf{WLP}} : \textsc{Stmt} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$, the consistent lifting of WLP. For

$$\widetilde{\mathsf{WLP}}(\texttt{skip}, \widetilde{\phi}) = \widetilde{\phi} \qquad \widetilde{\mathsf{WLP}}(s_1\texttt{;}\ s_2, \widetilde{\phi}) = \widetilde{\mathsf{WLP}}(s_1, \widetilde{\mathsf{WLP}}(s_2, \widetilde{\phi}))$$

$$\widetilde{\mathsf{WLP}}(x\ \texttt{:=}\ e, \widetilde{\phi}) = \widetilde{\phi}[e/x] \qquad \widetilde{\mathsf{WLP}}(\texttt{assert}\ \phi_a, \widetilde{\phi}) = \phi_a\ \wedge\ \widetilde{\phi}$$

$$\widetilde{\mathsf{WLP}}(y\ \texttt{:=}\ m\texttt{(}x\texttt{)}, \widetilde{\phi}) = \begin{cases} \phi' & \text{if } \widetilde{\phi}, \mathsf{mpre}(m), \mathsf{mpost}(m) \in \textsc{Formula} \\ \phi'\ \wedge\ \texttt{?} & \text{otherwise} \end{cases}$$

$$\text{where } \phi' = \max_{\Rightarrow} \{\ \phi'' \mid y \notin \mathsf{FV}(\phi'')\ \wedge\ (\phi'' \mathrel{\widetilde{\Rightarrow}} \mathsf{mpre}(m)[x/\mathsf{mparam}(m)]) \wedge$$
$$(\phi''\ \wedge\ \mathsf{mpost}(m)[x, y/\texttt{old(}\mathsf{mparam}(m)\texttt{)}, \texttt{result}]) \mathrel{\widetilde{\Rightarrow}} \widetilde{\phi}\ \}$$

Fig. 5. GVL: Weakest precondition (selected rules)

most statements $\widetilde{\mathsf{WLP}}$ is defined almost identical to WLP, however, calls are more complex. Note that for calls, $\widetilde{\mathsf{WLP}}$ not only has to deal with the fact that $\widetilde{\phi}$ is a gradual formula, but also that procedure m may now have imprecise contracts. In a sense, the function is lifted w.r.t. three formula parameters, two of them referenced through the procedure's name. To accomplish this, we first determine the static part ϕ' of the result which is analogous to the WLP, but resorting to lifted predicates. Next, we determine whether it would be sufficient to return ϕ' unmodified, or whether it is plausible that the precondition must be stronger. If all three influencing formulas are precise $\widetilde{\mathsf{WLP}}$ should coincide with WLP, so ϕ' is returned. Otherwise, ϕ' might have been chosen too weak, which is counteracted by making it imprecise.

Abstract interpretation. Again, AGT [7] formalizes the notion of consistent functions using an *abstraction* function α that maps a set of static formulas back to the most precise gradual formula that represents this set, such that $\langle \gamma, \alpha \rangle$ forms a Galois connection.

Definition 12 (Abstraction of formulas). *Let* $\alpha : \mathcal{P}^{\textsc{SatFormula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *be defined as* $\alpha(\overline{\phi}) = \min_{\sqsubseteq} \{\ \widetilde{\phi} \in \widetilde{\textsc{Formula}} \mid \overline{\phi} \subseteq \gamma(\widetilde{\phi})\ \}$

α is partial since $\min_{\sqsubseteq}$ does not necessarily exist, *e.g.* $\alpha(\{\texttt{(x}\ \neq\ \texttt{x)}, \texttt{(x = x)}\})$ is undefined. Using concretization for gradual parameters and abstraction for return values one can consistently lift (partial) functions:

Definition 13 (Consistent Function Lifting). *Given a partial function* $f : \textsc{Formula} \rightharpoonup \textsc{Formula}$*, its consistent lifting* $\widetilde{f} : \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *is defined as* $\widetilde{f}(\widetilde{\phi}) = \alpha(\{\ f(\phi) \mid \phi \in \gamma(\widetilde{\phi})\ \})$

Lemma 2 (Consistent WLP). $\widetilde{\mathsf{WLP}}$ *is a consistent lifting of* WLP*.*

3.4 Lifting the Verification Judgment

Gradual verification in GVL must deal with imprecise contracts. The static verifier of SVL uses WLP and implication to determine whether contracts and the overall program are valid (Def. 4, 5). Because contracts in GVL may be imprecise, we have to resort to $\widetilde{\mathsf{WLP}}$ (Fig. 5) and consistent implication (Def. 10).

Example 3 (Static Checker of GVL). Static semantics of SVL and GVL coincide for precise contracts, so example 2 applies to GVL without modification. We extend the example with imprecise contracts:

requires (balance ≥ amount) ensures (result ≥ 0) ∧ ?

Note the similarity to the precise contract in example 1 which causes GVL's static checker to reject the main procedure. However, with the imprecise postcondition we now have `(balance ≥ 0) ∧ ?` $\widetilde{\Rightarrow}$ `(balance ≥ 40)`. As a result, the static checker optimistically accepts the program. At the same time, it is not *guaranteed* that the precondition is satisfied at runtime without additional checks. We expect GVL's runtime semantics (Section 4) to add such checks as appropriate. These runtime checks should succeed for the main procedure of example 1, however they should fail if we modify the main program as follows, withdrawing more money than available:

```
int b := 100; b := withdraw(b, 30); b := withdraw(b, 80);
```

Static checking succeeds since `(b ≥ 0) ∧ ?` $\widetilde{\Rightarrow}$ `(b ≥ 80)`, but `b`'s value at runtime will not satisfy the formula. Note that the presence of imprecision does not necessarily imply success of static checking:

```
int b := 100; b := withdraw(b, 30); assert (b < 0);
```

It is *implausible* that this program is valid since `(b ≥ 0) ∧ ?` $\widetilde{\Rightarrow}$ `(b < 0)` does not hold. However, further weakening `withdraw`'s postcondition to `?` would again result in static acceptance but dynamic rejection.

requires ? ensures (result = old(balance) - old(amount)) ∧ ?

This contract demonstrates that imprecision must not necessarily result in runtime checks. The body's $\widetilde{\mathsf{WLP}}$ is `?`, which is implied by the annotated precondition `?` without having to be optimistic (i.e. resort to the plausibility interpretation). We expect that an efficient runtime semantics, like the one we discuss in Section 4.3, adds no runtime overhead through checks here.

4 GVL: Dynamic Semantics

Accepting a gradually-verified program means that it is *plausible* that the program remains valid during each step of its execution, precisely as it is guaranteed by soundness of SVL. To prevent a GVL program from stepping from a valid

state into an invalid state, we extend the dynamic semantics of SVL with (desirably minimal) runtime checks. As soon as the validity of the execution is no longer plausible, these checks cause the program to step into a dedicated error state. Example 3 motivates this behavior.

Soundness. We revise the soundness definition of SVL to the gradual setting which will guide the upcoming efforts to achieve soundness via runtime assertion checks. Validity of states (Def. 6) relies on sWLP (Fig. 4) which itself consumes postconditions of procedures. Hence, GVL uses a consistent lifting of sWLP which we define analogous to Fig. 4, but based on $\widetilde{\mathsf{WLP}}$. Save for using $\mathsf{s}\widetilde{\mathsf{WLP}}$ instead of sWLP, valid states of GVL are defined just like those of SVL.

We expect there to be error derivations $\pi \mathrel{\widetilde{\longrightarrow}} \textbf{error}$ whenever it becomes implausible that the remaining program can be run safely. Note that we do not extend State, but instead define $\cdot \mathrel{\widetilde{\longrightarrow}} \cdot \subseteq \textsc{State} \times (\textsc{State} \cup \{\textbf{error}\})$. As a result, we can leave Prop. 2 (Preservation) untouched.

In Section 4.1 we derive a naive runtime semantics driven by the soundness criteria of GVL. We then examine the properties of the resulting gradually verified language. In Section 4.3 we discuss optimizing this approach by combining $\widetilde{\mathsf{WLP}}$ with strongest preconditions $\widetilde{\mathsf{SP}}$ in order to determine statically-guaranteed information that can be used to minimize the runtime checks ahead of time.

4.1 Naive semantics

We start with a trivially correct but expensive strategy of adding runtime assertions to each execution step, checking whether the new state would be valid (preservation), right before actually transitioning into that state (progress). [6]

Let $\rho'_{1..m}, \rho_{1..n} \in \textsc{Env},\ s'_{1..m}, s_{1..n} \in \textsc{Stmt}$

If $\langle \rho'_m, s'_m \rangle \cdot ... \cdot \langle \rho'_1, s'_1 \rangle \cdot \mathsf{nil} \longrightarrow \langle \rho_n, s_n \rangle \cdot ... \cdot \langle \rho_1, s_1 \rangle \cdot \mathsf{nil}$ holds[7], then

$$\langle \rho'_m, s'_m \rangle \cdot ... \cdot \langle \rho'_1, s'_1 \rangle \cdot \mathsf{nil} \mathrel{\widetilde{\longrightarrow}} \begin{cases} \langle \rho_n, s_n \rangle \cdot ... \cdot \langle \rho_1, s_1 \rangle \cdot \mathsf{nil} & \text{if } (\rho_n \mathrel{\widetilde{\vDash}} \widetilde{\phi}_n) \wedge ... \wedge (\rho_1 \mathrel{\widetilde{\vDash}} \widetilde{\phi}_1) \\ \quad \text{where } \widetilde{\phi}_n \cdot ... \cdot \widetilde{\phi}_1 \cdot \mathsf{nil} = \mathsf{s}\widetilde{\mathsf{WLP}}(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true}) & \\ \textbf{error} & \text{otherwise} \end{cases}$$

Before showing how to implement the above semantics, we confirm its soundness: Progress of GVL follows from progress of SVL. The same is true for preservation: in the first reduction case, validity of the resulting state follows from preservation of SVL.

[6] Note the difference between runtime assertions and the `assert` statement. The former checks assertions at runtime, transitioning into a dedicated exceptional state on failure. The latter is a construct of a statically verified language, and is hence implementable as a no-operation.

[7] SsCall and SsCallExit as defined in Fig. 2 are not defined for gradual formulas. Thus, we adjust those rules to use consistent evaluation $\widetilde{\vDash}$ instead of $\vDash$. Since $\widetilde{\vDash}$ coincides with $\vDash$ for precise formulas, this is a conservative extension of SVL.

While we can draw the implementation of $\cdot \widetilde{\longrightarrow} \cdot$ from SVL, implementing the case condition $(\rho_n \mathrel{\widetilde{\vDash}} \widetilde{\phi}_n) \wedge ... \wedge (\rho_1 \mathrel{\widetilde{\vDash}} \widetilde{\phi}_1)$ results in overhead. As a first step, we can heavily simplify this check using inductive properties of our language: Stack frames besides the topmost one are not changed by a single reduction, *i.e.* $\rho_{n-1}, ..., \rho_1, s_{n-1}, ..., s_1$ stay untouched. It follows that $\widetilde{\phi}_i$ for $1 \leq i < n$ remains unchanged since changes in s_n do not affect lower components of $\mathsf{s}\widetilde{\mathsf{WLP}}$ (see Fig. 4). As a result, it is sufficient to check $\rho_n \mathrel{\widetilde{\vDash}} \mathsf{s}\widetilde{\mathsf{WLP}}_n(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$.

Recall how we argued that a weakest precondition approach is more suited for the dynamic semantics of GVL than Hoare logic. Due to the syntax-directed sequence rule, all potentially occurring $\mathsf{s}\widetilde{\mathsf{WLP}}_n$ are partial results of statically *precomputed* preconditions. Contrast this with a gradual sequence rule of Hoare logic: {?}`skip; skip`{?} could be accepted statically by, say, instantiating the existential with `(x = 3)`, which is allowed if both premises of the rule are lifted independently. However, the partial result {`(x = 3)`}`skip`{?} has no (guaranteed) relationship with the next program state since the existential was chosen too strong. Any attempt to fix the gradual sequence rule by imposing additional restrictions on the existential must necessarily involve a weakest precondition calculus, applied to the suffix of the sequence.

4.2 Properties of GVL

Before discussing practical aspects of GVL, we turn to its formal properties: GVL is a *sound*, *gradual* language. The following three properties are formalized and proven in Coq.

Soundness. Our notion of soundness for GVL coincides with that of SVL, save for the possibility of runtime errors. Indeed, it is up to the dynamic semantics of GVL to make up for the imprecisions that weaken the statics of GVL.

Lemma 3 (Soundness of GVL). *GVL is sound:*

Progress *If* $\pi \in \textsc{State}$ *is a valid state and* $\pi \notin \{\langle \rho, \mathtt{skip} \rangle \cdot \mathsf{nil} \mid \rho \in \textsc{Env}\}$ *then* $\pi \widetilde{\longrightarrow} \pi'$ *for some* $\pi' \in \textsc{State}$ *or* $\pi \widetilde{\longrightarrow}$ **error**.

Preservation *If* π *is a valid state and* $\pi \widetilde{\longrightarrow} \pi'$ *for some* $\pi' \in \textsc{State}$ *then* π' *is a valid state.*

We call the state $\langle \rho_n, s_n \rangle \cdot ... \cdot \langle \rho_1, s_1 \rangle \cdot \mathsf{nil}$ valid *if* $\rho_i \mathrel{\widetilde{\vDash}} \mathsf{s}\widetilde{\mathsf{WLP}}_i(s_n \cdot ... \cdot s_1 \cdot \mathsf{nil}, \mathtt{true})$ *for all* $1 \leq i \leq n$.

Conservative extension. GVL is a conservative extension of SVL, meaning that both languages coincide on fully-precise programs. This property is true *by construction.* Indeed, the definition of concretization and consistent lifting captures this property, which thus percolates to the entire verification logic. In order for the dynamic semantics to be a conservative extension, GVL must progress whenever SVL does, yielding the same continuation. This is the case since the reduction rules of GVL coincide with those of SVL for fully-precise annotations (the runtime checks succeed due to preservation of SVL, so we do not step to **error**).

Gradual guarantees. Siek *et al.* formalize a number of criteria for gradually-typed languages [19], which we can adapt to the setting of program verification. In particular, the *gradual guarantee* captures the smooth continuum between static and dynamic verification. More precisely, it states that typeability (here, verifiability) and reducibility are *monotone* with respect to precision. We say a program p_1 is more precise than program p_2 ($p_1 \sqsubseteq p_2$) if p_1 and p_2 are equivalent except in terms of contracts and if p_1's contracts are more precise than p_2's contracts. A contract `requires` ϕ_p^1 `ensures` ϕ_q^1 is more precise than contract `requires` ϕ_p^2 `ensures` ϕ_q^2 if $\phi_p^1 \sqsubseteq \phi_p^2$ and $\phi_q^1 \sqsubseteq \phi_q^2$.

In particular, the static gradual guarantee for verification states that a valid gradual program is still valid when we reduce the precision of contracts.

Proposition 3 (Static gradual guarantee of verification).
Let $p_1, p_2 \in \textsc{Program}$ *such that* $p_1 \sqsubseteq p_2$*. If* p_1 *is valid then* p_2 *is valid.*

The dynamic gradual guarantee states that a valid program that takes a step still takes the same step if we reduce the precision of contracts.

Proposition 4 (Dynamic gradual guarantee of verification).
Let $p_1, p_2 \in \textsc{Program}$ *such that* $p_1 \sqsubseteq p_2$ *and* $\pi \in \textsc{State}$*.*
If $\pi \widetilde{\longrightarrow}_{p_1} \pi'$ *for some* $\pi' \in \textsc{State}$ *then* $\pi \widetilde{\longrightarrow}_{p_2} \pi'$*.*

This also means that if a gradual program fails at runtime, then making its contracts more precise will *not* eliminate the error. In fact, doing so may only make the error manifest *earlier* during runtime or manifest *statically.* This is a fundamental property of gradual verification: a runtime verification error reveals a fundamental mismatch between the gradual program and the underlying verification discipline.

4.3 Practical aspects

Residual checks. Compared to SVL, the naive semantics adds a runtime assertion to every single reduction. Assuming that the cost of checking an assertion is proportional to the formula size, *i.e.* proportional to the size of the WLP of the remaining statement, this is highly unsatisfying. The situation is even worse if the entire GVL program has fully-precise annotations, because then the checks are performed even though they are not necessary for safety.

We can reduce formula sizes given static information, expecting formulas to vanish (reduce to `true`) in the presence of fully-precise contracts and gradually grow with the amount of imprecision introduced, yielding a pay-as-you-go cost model. Example 4 illustrates this relationship.

Example 4 (Residual checks). Consider the following variation of `withdraw`:

```
int withdraw(int balance, int amount)
      requires ? ensures result ≥ 0 {
   // WLP: (balance - amount ≥ 0) ∧ (amount > 0)
   assert amount > 0;
```

```
    // WLP: balance - amount ≥ 0
    balance = balance - amount;
    // WLP: balance ≥ 0
    return balance;
    // WLP: result ≥ 0
}
```

Precomputed preconditions are annotated. Following the naive semantics (Section 4.1), these are to be checked *before* entering the corresponding state, to ensure soundness. However, many of these checks are redundant. When entering the procedure (*i.e.* stepping to the state prior to the assertion statement), we must first check $\phi_1 =$ `(balance - amount ≥ 0) ∧ (amount > 0)`. According to the naive semantics, in order to execute the assertion statement, we would then check $\phi_2 =$ `(balance - amount ≥ 0)`. Fortunately, it is derivable statically that this check must definitely succeed: The strongest postcondition of `assert amount > 0` given ϕ_1 is again ϕ_1. Since $\phi_1 \Rightarrow \phi_2$ there is no point in checking ϕ_2 at runtime. Similar reasoning applies to both remaining statements, making all remaining checks redundant. Only the initial check remains, which is itself directly dependent on the imprecision of the precondition. Preconditions `(balance - amount ≥ 0) ∧ ?` or `(amount > 0) ∧ ?` would allow dropping the corresponding term of the formula, the conjunction of both (with or without a `?`) allows dropping the entire check.

The above example motivates the formalization of a strongest postcondition function $\widetilde{\mathsf{SP}}$ and function $\widetilde{\mathsf{diff}}$ which takes a formula $\widetilde{\phi}_a$ to check, a formula $\widetilde{\phi}_k$ known to be true and calculates a residual formula $\widetilde{\mathsf{diff}}(\widetilde{\phi}_a, \widetilde{\phi}_k)$ sufficient to check in order to guarantee that $\widetilde{\phi}_a$ holds.

Definition 14 (Strongest postcondition). *Let* $\mathsf{SP} : \textsc{Stmt} \times \textsc{Formula} \rightharpoonup \textsc{Formula}$ *be defined as:* $\mathsf{SP}(s, \phi) = \min_{\Rightarrow} \{\ \phi' \in \textsc{Formula} \mid \phi \Rightarrow \mathsf{WLP}(s, \phi')\ \}$

Furthermore let $\widetilde{\mathsf{SP}} : \textsc{Stmt} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *be the consistent lifting (Def. 13) of* SP.

Definition 15 (Reducing formulas).
Let $\mathsf{diff} : \textsc{Formula} \times \textsc{Formula} \rightharpoonup \textsc{Formula}$ *be defined as:*

$$\mathsf{diff}(\phi_a, \phi_k) = \max_{\Rightarrow} \{\ \phi \in \textsc{Formula} \mid (\phi\ \wedge\ \phi_k \Rightarrow \phi_a)\ \wedge\ (\phi\ \wedge\ \phi_k \in \textsc{SatFormula})\ \}$$

Furthermore let $\widetilde{\mathsf{diff}} : \widetilde{\textsc{Formula}} \times \widetilde{\textsc{Formula}} \rightharpoonup \widetilde{\textsc{Formula}}$ *be defined as:*

$$\widetilde{\mathsf{diff}}(\phi_a, \widetilde{\phi}_k) = \mathsf{diff}(\phi_a, \mathsf{static}(\widetilde{\phi}_k)) \qquad \widetilde{\mathsf{diff}}(\phi_a\ \wedge\ ?, \widetilde{\phi}_k) = \mathsf{diff}(\phi_a, \mathsf{static}(\widetilde{\phi}_k))\ \wedge\ ?$$

Both SP and diff can be implemented approximately by only approximating min/max. Likewise, an implementation of $\widetilde{\mathsf{SP}}$ may err towards imprecision. As a result, the residual formulas may be larger than necessary. [8]

[8] Even a worst case implementation of $\widetilde{\mathsf{SP}}(s, \widetilde{\phi})$ as ? will only result in no reduction of the checks, but not affect soundness.

$$\frac{\langle \rho'_n, (s;\ s_n)\rangle \cdot ... \longrightarrow \langle \rho_n, s_n\rangle \cdot ...}{\langle \rho'_n, (s;\ s_n)\rangle \cdot ... \widetilde{\longrightarrow} \langle \rho_n, s_n\rangle \cdot ...}\ \widetilde{\text{SsLocal}}$$

$$\frac{\begin{array}{c}\langle \rho_{n-1}, (y := m(x);\ s_{n-1})\rangle \cdot ... \longrightarrow \langle \rho_n, \mathsf{mbody}(m)\rangle \cdot ... \\ \rho_n \mathrel{\widetilde{\vDash}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{WLP}}(\mathsf{mbody}(m), \mathsf{mpost}(m)), \mathsf{mpre}(m))\end{array}}{\langle \rho_{n-1}, (y := m(x);\ s_{n-1})\rangle \cdot ... \widetilde{\longrightarrow} \langle \rho_n, \mathsf{mbody}(m)\rangle \cdot ...}\ \widetilde{\text{SsCall}}$$

$$\frac{\begin{array}{l}\langle \rho'_{n+1}, \mathtt{skip}\rangle \cdot \langle \rho'_n, (y := m(x);\ s_n)\rangle \cdot ... \longrightarrow \langle \rho_n, s_n\rangle \cdot ... \\ \rho_n \mathrel{\widetilde{\vDash}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{sWLP}}_n(s_n \cdot ..., \mathtt{true}), \\ \qquad \widetilde{\mathsf{SP}}(y := m(x), \widetilde{\mathsf{sWLP}}_n((y := m(x);\ s_n) \cdot ..., \mathtt{true})))\end{array}}{\langle \rho'_{n+1}, \mathtt{skip}\rangle \cdot \langle \rho'_n, (y := m(x);\ s_n)\rangle \cdot ... \widetilde{\longrightarrow} \langle \rho_n, s_n\rangle \cdot ...}\ \widetilde{\text{SsCallExit}}$$

Fig. 6. Dynamic semantics with reduced checks.

Definition 16 (Approximate algorithm for $\widetilde{\mathsf{diff}}$).

```
F̃ORMULA d̃iff (F̃ORMULA φ̃_a , F̃ORMULA φ̃_b)
  let φ̃_1 ∧ φ̃_2 ∧ ... ∧ φ̃_n := φ̃_a (such that all φ̃_i are atomic)
  for i from 1 to n
    if ⟦φ̃_b⟧ ∩ ⟦φ̃_1 ∧ ... ∧ φ̃_i-1 ∧ φ̃_i+1 ... ∧ φ̃_n⟧ ⊆ ⟦φ̃_a⟧
      φ̃_i := true
  return φ̃_1 ∧ ... ∧ φ̃_n        // one may drop the true terms
```

Figure 6 shows the dynamic semantics using residual checks (omitting error reductions). Runtime checks of reductions operating on a single stack frame vanish completely as there exists no source of imprecision in that subset of GVL. This property can be formalized as: *For all $s \in$ STMT that do not contain a call,* $\widetilde{\mathsf{diff}}(\widetilde{\mathsf{sWLP}}_n(s_n \cdot ..., \mathtt{true}), \widetilde{\mathsf{SP}}(s, \widetilde{\mathsf{sWLP}}_n((s;\ s_n) \cdot ..., \mathtt{true}))) \in \{\mathtt{true}, \mathtt{?}\}$ The check in $\widetilde{\text{SsCall}}$ vanishes if $\mathsf{mpre}(m)$ is precise. The check in $\widetilde{\text{SsCallExit}}$ vanishes if $\mathsf{mpost}(m)$ is precise. Recall that Example 4 demonstrates this effect: We concluded that only the initial check is necessary and derived that it also vanishes if the precondition is precise.

If $\mathsf{mpost}(m)$ is imprecise, the assertion is equivalent to

$$\rho_n \mathrel{\widetilde{\vDash}} \widetilde{\mathsf{diff}}(\widetilde{\mathsf{diff}}(\widetilde{\mathsf{sWLP}}_n(s_n \cdot ..., \mathtt{true}), \mathsf{mpre}(m)[x/\mathsf{mparam}(m)]), \\ \mathsf{mpost}(m)[x, y/\mathtt{old}(\mathsf{mparam}(m)), \mathtt{result}])$$

which exemplifies the pay-as-you-go relationship between level of imprecision and run-time overhead: both $\mathsf{mpre}(m)$ and $\mathsf{mpost}(m)$ contribute to the reduction of $\widetilde{\mathsf{sWLP}}_n(s_n \cdot ..., \mathtt{true})$, *i.e.* increasing their precision results in smaller residual checks.

Pay-as-you-go. To formally establish the pay-as-you-go characteristic of gradual verification, we introduce a simple cost model for runtime contract checking.

Let $\mathsf{size}(\widetilde{\phi})$ be the number of conjunctive terms of (the static part of) $\widetilde{\phi}$, e.g. $\mathsf{size}((\mathtt{x} = 3) \wedge (\mathtt{y} \neq \mathtt{z}) \wedge \mathtt{?}) = 2$. We assume that measure to be proportional to the time needed to evaluate a given formula. Let $\mathsf{checksize}(p)$ be the sum of the size of all residual checks in program p.

Proposition 5 (Pay-as-you-go overhead).
a) *Given programs* p_1, p_2 *such that* $p_1 \sqsubseteq p_2$*, then* $\mathsf{checksize}(p_1) \leq \mathsf{checksize}(p_2)$*.*
b) *If a program* p *has only* precise *contracts, then* $\mathsf{checksize}(p) = 0$*.*

5 Scaling up to Implicit Dynamic Frames

We applied our approach to a richer program logic, namely *implicit dynamic frames* (IDF) [20], which enables reasoning about shared mutable state. The starting point is an extended statically verified language similar to Chalice [10], called SVL_{IDF}. Compared to SVL, the language includes classes with publicly-accessible fields and instance methods. We add field assignments and object creation. Formulas may also contain field *accessibility predicates* `acc` from IDF and use the separating conjunction `*` instead of regular (non-separating) conjunction $\wedge$. An accessibility predicate $\mathtt{acc}(e.f)$ denotes exclusive access to the field $e.f$. It justifies accessing $e.f$ both in the source code (*e.g.* `x.f := 3` or `y := x.f`) and in the formula itself (*e.g.* $\mathtt{acc(x.f)} * \mathtt{(x.f} \neq \mathtt{4)}$), which is called *framing.* Compared to SVL, the main challenge in gradualizing SVL_{IDF} is framing.

The linear logic ensures that accessibility predicates cannot be duplicated, hence entitling them to represent *exclusive* access to a field. SVL_{IDF} can *statically* prove that any field access during execution is justified. To formalize and prove soundness, SVL_{IDF} has a reference dynamic semantics that tracks, for each stack frame, the set of fields that it has exclusive access to; deciding at runtime whether a formula holds depends on this information. Of course, thanks to soundness, an implementation of SVL_{IDF} needs no runtime tracking.

Recall that our approach to derive a gradual language includes the insertion of runtime checks, which requires that formulas *can* be evaluated at runtime. Therefore, the overhead of the reference semantics of SVL_{IDF} carries over to a naive implementation semantics. Additionally, it is no longer clear how accessibility is split between stack frames in case of a call: SVL_{IDF} transfers exclusive access to fields that are mentioned in the precondition of a procedure from the call site to the callee's (fresh) stack frame. As we allow `?` to also plausibly represent accessibility predicates, an imprecise precondition poses a challenge.

A valid strategy is to conservatively forward *all* accesses from caller to callee. As for GVL, we can devise an efficient implementation strategy for accessibility tracking that results in pay-as-you-go overhead.The fact that reducing the precision of contracts may now result in a divergence of program states (specifically, the accessible fields) asks for an adjustment of the dynamic part of the gradual guarantee, which originally requires lock-step reduction behavior. We carefully adjust the definition, preserving the core idea that reducing precision of a valid program does not alter the *observable* behavior of that program. The formalization of SVL_{IDF} and GVL_{IDF} are available in a companion

report available online, along with an interactive prototype implementation at `http://olydis.github.io/GradVer/impl/HTML5wp/`.

The prototype implementation of $\mathrm{GVL}_{\mathrm{IDF}}$ displays the static and dynamic semantics of code snippets interactively, indicating the location of inserted runtime checks where necessary. It also displays the stack and heap at any point of execution. A number of predefined examples are available, along with an editable scratchpad. In particular, Example 2 demonstrates how imprecision enables safe reasoning about a recursive data structure that was not possible in $\mathrm{SVL}_{\mathrm{IDF}}$, because $\mathrm{SVL}_{\mathrm{IDF}}$ lacks recursive predicates. This illustrates that, similarly to gradual types, imprecision can be used to augment the expressiveness of the static verification logic. In this case, the example does not even require a single runtime check.

6 Related Work

We have already related our work to the most-closely related research, including work on the underlying logics [8,4,15,20], the theory of gradual typing [18,17,19,7], closely related approaches to static [10] and dynamic [6,12,11,3] verification, as well as hybrid verification approaches [2,14]. Additional related work includes gradual type systems that include notions of *ownership* or *linearity*; one can think of the `acc` predicate as representing ownership of a piece of the heap, and `acc` predicates are treated linearly in the implicit dynamic frames methodology [20]. [21] developed a gradual typestate checking system, in which the state of objects is tracked in a linear type system. Similar to `acc` predicates, permissions to objects with state are passed linearly from one function to another, without being duplicated; if a strong permission is lost (*e.g.* due to a gradual specification), it can be regained with a runtime check, as long as no conflicting permission exist.

The gradual ownership approach of [16] is also related in that, like implicit dynamic frames, it aids developers in reasoning (gradually) about heap data structures. In that work, developers can specify containment relationships between objects, such that an *owned* object cannot be accessed from outside its owner. These access constraints can be checked either statically using a standard ownership type system, but if the developer leaves out ownership annotations from part of the program, dynamic checks are inserted.

Typestate and ownership are finite-state and topological properties, respectively, whereas in this work we explore gradual specification of logical contracts for the first time. Neither of these prior efforts benefited from the Abstracting Gradual Typing (AGT) methodology [7], which guided more principled design choices in our present work. Additionally, it is unclear whether the gradual guarantee of Siek *et al.* [19] holds in these proposals, which were developed prior to the formulation of the gradual guarantee.

One contrasting effort, which was also a stepping-stone to our current paper, is recent work on gradual refinement types [9]. In that approach, the AGT methodology is applied to a functional language in which types can be refined by

logical predicates drawn from a decidable logic. The present work is in a different context, namely first-order imperative programs as opposed to higher-order pure functional programs. This difference has a strong impact on the technical development. The present work is simpler in one respect, because formulas do not depend on their lexical context as in the gradual refinement setting. However, we had to reformulate gradual verification based on a weakest precondition calculus, whereas the prior work could simply extend the type system setting used when the AGT methodology was proposed. In addition, we provide a runtime semantics directly designed for the gradual verification setting, rather than adapting the evidence-tracking approach set forth by the AGT methodology and used for gradual refinement types. In fact, we investigated using the evidence-based approach for the runtime semantics of gradual verification, and found that it was semantically equivalent to what we present here but introduces unnecessary complexities. Overall, by adapting the AGT methodology to the verification setting, this work shows that the abstract interpretation approach to designing gradual languages can scale beyond type systems.

7 Conclusion

We have developed the formal foundations of gradual program verification, taking as starting point a simple program logic. This work is the first adaptation of recent fundamental techniques for gradual typing to the context of program verification. We have shown how to exploit the Abstracting Gradual Typing methodology [7] to systematically derive a gradual version of a weakest precondition calculus. Gradual verification provides a continuum between static and dynamic verification techniques, controlled by the (im)precision of program annotations; soundness is mediated through runtime checks.

Later, we briefly discuss how we applied our strategy to a more advanced logic using implicit dynamic frames (IDF) [20] in order to reason about mutable state. The use of IDF presents an additional challenge for obtaining a full pay-as-you-go model for gradual verification, because the footprint has to be tracked. We point to our prototype implementation which also references a formalization of graualizing $\mathrm{SVL}_{\mathrm{IDF}}$. Further optimization of this runtime tracking is an interesting direction of future work. Another interesting challenge is to exercise our approach with other, richer program logics, as well as to study the gradualization of type systems that embed logical information, such as Hoare Type Theory [13], establishing a bridge between this work and gradual refinement types [9].

References

1. Arlt, S., Rubio-González, C., Rümmer, P., Schäf, M., Shankar, N.: The gradual verifier. In: NASA Formal Methods Symposium. pp. 313–327. Springer (2014)
2. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of jml tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005), http://dx.doi.org/10.1007/s10009-004-0167-4
3. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the java modeling language (jml) (2002)
4. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (Aug 1975), http://doi.acm.org/10.1145/360933.360975
5. Fahndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: ACM SAC - OOPS. Association for Computing Machinery, Inc. (March 2010), https://www.microsoft.com/en-us/research/publication/embedded-contract-languages/
6. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2002). pp. 48–59. Pittsburgh, PA, USA (Sep 2002)
7. Garcia, R., Clark, A.M., Tanter, E.: Abstracting gradual typing. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 429–442. POPL '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2837614.2837670
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
9. Lehmann, N., Tanter, É.: Gradual refinement types. In: Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017). pp. 775–788. Paris, France (Jan 2017)
10. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with chalice. In: Foundations of Security Analysis and Design V, pp. 195–222. Springer (2009)
11. Meyer, B.: Eiffel: A language and environment for software engineering. Journal of Systems and Software 8(3), 199–246 (1988)
12. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
13. Nanevski, A., Morrisset, G., Birkedal, L.: Hoare type theory, polymorphism and separation. Journal of Functional Programming 5-6, 865–911 (2008)
14. Nguyen, H.H., Kuncak, V., Chin, W.N.: Runtime checking for separation logic. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 203–217. Springer (2008)
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. pp. 55–74. IEEE (2002)
16. Sergey, I., Clarke, D.: Gradual ownership types. In: Proceedings of the 21st European Conference on Programming Languages and Systems. pp. 579–599. ESOP'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-28869-2_29
17. Siek, J., Taha, W.: Gradual typing for objects. In: European Conference on Object-Oriented Programming. pp. 2–27. Springer (2007)

18. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. vol. 6, pp. 81–92 (2006)
19. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
20. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: European Conference on Object-Oriented Programming. pp. 148–172. Springer (2009)
21. Wolff, R., Garcia, R., Tanter, É., Aldrich, J.: Gradual typestate. In: European Conference on Object-Oriented Programming. pp. 459–483. Springer (2011)
22. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (Nov 1994), http://dx.doi.org/10.1006/inco.1994.1093

Automatic Verification of RMA Programs via Abstraction Extrapolation

Cedric Baumann[1], Andrei Marian Dan[1], Yuri Meshman[2], Torsten Hoefler[1], and Martin Vechev[1]

[1] Department of Computer Science, ETH Zurich, Switzerland
[2] IMDEA Software Institute, Madrid, Spain

Abstract. Remote Memory Access (RMA) networks are emerging as a promising basis for building performant large-scale systems such as MapReduce, scientific computing applications, and others. To achieve this performance, RMA networks exhibit relaxed memory consistency. This means the developer now must manually ensure that the additional relaxed behaviors are not harmful to their application – a task known to be difficult and error-prone. In this paper, we present a method and a system that can automatically address this task. Our approach consists of two ingredients: (i) a *reduction* where we reduce the task of verifying program P running on RMA to the problem of verifying a program $\overline{P}$ on sequential consistency (where $\overline{P}$ captures the required RMA behaviors), and (ii) *abstraction extrapolation*: a new method to automatically discover both, predicates (via *predicate extrapolation*) and abstract transformers (via *boolean program extrapolation*) for $\overline{P}$. This enables us to automatically *extrapolate* the proof of P under sequential consistency (SC) to a proof of P under RMA. We implemented our method and showed it to be effective in automatically verifying, for the first time, several challenging concurrent algorithms under RMA.

1 Introduction

Remote Memory Access (RMA) programming is a technology used in modern data centers to communicate between machines with low overhead. It enables low latencies ($< 1\mu s$ [26]) and high bandwidth. In RMA, remote operations are executed by the underlying network interface controller bypassing the CPU and the operating system (in contrast to regular network operations). The network card reads the data using Direct Memory Access (DMA), sends it over the network, and finally the receiving network card writes the data using DMA. This approach is faster than traditional network communication because in data centers, the direct access propagation delay is small compared to a network message stack overhead of standard sockets. RMA technology is available in several networks: InfiniBand [43], IBM Blue Gene Q [6], IBM PERCS [9], Cray Gemini [7] and Aries [25]. Typically, the RMA functionality is available through RMA libraries (InfiniBands OFED [37], Cray DMAPP [18], Portals4 [12]). Middleware applications, such as the Hadoop File System [32], then call the RMA interface directly.

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 47–70, 2018.
https://doi.org/10.1007/978-3-319-73721-8_3

Newer developments introduce RMA extensions for Ethernet (RoCE [13]) or IP routable RMA protocols (iWARP [40]).

RMA instructions of a program are executed asynchronously, i.e., the execution of the program proceeds without waiting for the completion of the remote RMA operations. To offer guarantees on the completion of remote operations, RMA provides the `flush` statement, which enforces that all remote operations to a certain machine are executed before the execution continues.

As expected, verifying programs running under RMA is challenging because they exhibit additional relaxed behaviors beyond those allowed by sequential consistency (SC). Moreover, programs executing under RMA exhibit behaviors not possible under other relaxed memory models such as Total Store Order (x86 TSO) or Partial Store Order (PSO) [19]. The goal of this paper is to address this challenge, namely, develop automated techniques for verifying RMA programs.

The problem. Given a program P running on an RMA network and a safety specification S, our goal is to answer whether P satisfies S under RMA, indicated as $P \models_{RMA} S$.

Our Work. In our work we approach this challenge via predicate abstraction [28], a method shown effective in verifying concurrent programs [24, 30] and x86 TSO and PSO programs [20]. Standard predicate abstraction ([11, 28]) assumes sequential consistency (SC). Given a program P and a set of predicates $V = \{p_1, \ldots, p_n\}$ over the variables in P, standard predicate abstraction builds a boolean program $B = \mathcal{BP}(P, V)$ that contains one boolean variable for each predicate in V. The boolean program comes with the guarantee that if B satisfies a property S, then the program P satisfies S as well:

$$B \models_{SC} S \implies P \models_{SC} S$$

Checking whether $B \models_{SC} S$ is typically done via a (three-valued) model checker. If a spurious counter-example execution trace is found by the model checker, then the initial set of predicates V is refined and the procedure is repeated. An overview of this approach is illustrated in the left part of Figure 1.

Reduction. However, the above guarantee does not hold when replacing SC with relaxed memory models, such as x86 TSO or PSO, because naive application of standard predicate abstraction for relaxed memory model programs is unsound ([20]). To address this issue, we reduce the problem of verifying RMA programs to SC verification (the Reduction box in Figure 1) by automatically constructing a new program $\overline{P}$ that captures RMA behaviors as part of the program. $\overline{P}$ uses set variables and boolean flags to account for these behaviors. If $\overline{P}$ satisfies a property S under sequential consistency, then P satisfies S under RMA:

$$\overline{P} \models_{SC} S \implies P \models_{RMA} S$$

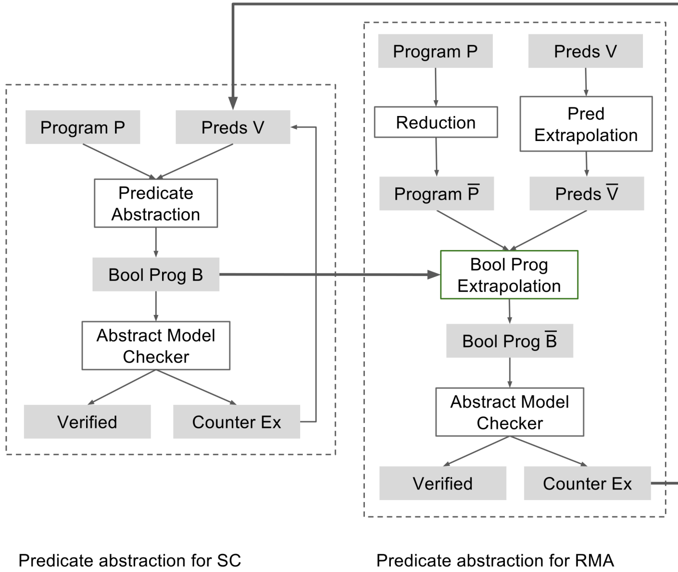


Fig. 1. Predicate abstraction for sequential consistency (left) and our verification method for RMA programs (right).

Predicate extrapolation. Given the newly generated program $\overline{P}$, we automatically generate a set of predicates $\overline{V}$ based on the original set of predicates V (the Pred Extrapolation box in Figure 1). This process is called *predicate extrapolation* and we denote it as $\mathcal{EP}(V) = \overline{V}$. A part of the new predicates in $\overline{V}$ contain universal quantifiers over elements of sets. Compared to the original set of predicates V, the extrapolated set $\overline{V}$ is approximatively twice as large (experimentally observed on our benchmarks), therefore limiting the applicability of standard predicate abstraction (requires an exponential number of calls to an SMT solver, in the worst case). We address this limitation by introducing the *boolean program extrapolation* technique.

Boolean program extrapolation. To side-step the potential exponential blow up, we construct a sound approximation of $\mathcal{BP}(\overline{\mathcal{P}}, \overline{\mathcal{V}})$: we introduce a novel extrapolation function $\mathcal{EBP}$ (the Bool Prog Extrapolation box in Figure 1) to construct a boolean program $\overline{B} = \mathcal{EBP}(\overline{P}, \overline{V}, B)$ without any call to the SMT solver. The *boolean program extrapolation* is based on the boolean program $B = \mathcal{BP}(P, V)$ and satisfies:

$$\overline{B} \models_{SC} S \implies \mathcal{BP}(\overline{P}, \overline{V}) \models_{SC} S$$

Overall approach. Automated verification of $P \models_{RMA} S$ takes place in four steps:

1. *Verify under SC*: given a set of predicates V, build a boolean program $B = \mathcal{BP}(P, V)$ and show that $B \models_{SC} S$.
2. *Reduce to SC*: Build the program $\overline{P}$ (to capture RMA behaviors for P) and extrapolate the corresponding set of predicates $\overline{V} = \mathcal{EP}(V)$ from the predicates V that worked under SC.
3. *Construct extrapolated boolean program*: given program $\overline{P}$, set of predicates $\overline{V}$, and boolean program $B = \mathcal{BP}(P, V)$, construct extrapolated boolean program $\overline{B} = \mathcal{EBP}(\overline{P}, \overline{V}, B)$.
4. *Verify boolean program*: whether $\overline{B} \models_{SC} S$, using a model checker. If the check fails due to abstraction imprecision, we refine the abstraction and restart the verification process. Otherwise, the program P satisfies property S under RMA and the process completes successfully.

Main Contributions. The main contributions of this paper are:

- A program reduction, based on sets, from RMA to SC. The construction allows us to handle traces with infinite number of relaxed memory operations.
- A novel abstraction approach for programs running on RMA, using predicates over sets and a *boolean program extrapolation* technique which requires no calls to the underlying SMT solver for building the boolean program.
- An implementation of our approach in a tool that can, for the first time, automatically verify several challenging (including infinite-state) concurrent algorithms running on the RMA model.

Our work can be seen as a step towards the more general problem of adapting the proof of one program to the proof of a more refined program, technically achieved here via abstraction extrapolation (in the case of predicate abstraction, extrapolation of the boolean program).

2 Overview

We illustrate our approach on a small RMA program shown in Figure 2. In this example, `Process` 1 declares a shared variable Y with initial value 1. `Process` 2 declares shared variables R and X, initializes them to 0 and 2, respectively. Next, it declares a local variable r. It then remotely puts the value of X into Y. Then, X is set to 3 and the value of Y is remotely read from `Process` 1 and is written to R. Finally, the process assigns R to local variable r. The specification (`assert`) is that at the end of the program (`final`), r is different than 3.

Under SC semantics, the only possible value of r at the end of the program is 2. However, under RMA semantics, r can have any value from the set {0,1,2,3}. Note that under other relaxed consistency models, such as x86 TSO, PSO, and C++ RMM, the value of 3 is not possible for r. Yet, under RMA, the `put` from the shared location X to Y, can be delayed until after assigning 3 to X. That value can then be read into R and put into local variable r, leading to an assertion violation. A developer will then have to manually infer the `flush` statements that are required to enforce the specification, when running under RMA semantics. We next introduce the semantics of RMA networks.

2.1 RMA Semantics

In RMA programs, a process can access shared variables of remote processes using remote statements such as `put` or `get`. These remote statements are executed *asynchronously* — the process executing them does not wait for the completion of the remote statement, instead it continues the execution of its program. In hardware, the program executing on a CPU relegates the remote operation execution to a component called Network Interface Controller (NIC) which connects to a NIC on a remote machine. The two NICs complete the operation, on shared locations assigned to the operation, without involving either the local or the remote CPU. `flush` statements are the main mechanism to guarantee that pending remote statements to a specific remote process are completed. A `flush`(pid) statement acts like a barrier, blocking the execution on the process until all pending remote statements to process pid are completed. The `flush` is expensive (increases latency) and should be used sparingly.

```
  Process 1:
1 shared Y = 1;

  Process 2:
1 shared R = 0, X = 2;
2 local r;
3 put(Y, 1, X);
4 store X = 3;
5 R = get(Y, 1);
6 load r = R;

  assert final (r != 3);
```

Fig. 2. RMA program consisting of two processes and a specification (shown at the end) which checks whether the value of local variable `r` is equal to `3`.

Syntax. We consider a basic programming language, shown in Table 1, that offers RMA primitives such as `put`, `get` and `flush`. The `load` u = X; statement reads the value of shared variable X and writes it in local variable u. The `store` X = expr statement writes to shared variable X the value of the expression on the right hand side (arithmetic expression over local variables). The `put` and `get` statements operate over shared variables X and Y, and also take as argument the identifier of the process storing the remote variable. The `flush` statement takes as argument a process id. The semantics of these statements are described next.

Table 1. Basic statements which capture the essence of RMA programming.

Statement	Description
`load` u = X;	local read
`store` X = expr;	local write
X = `get`(Y, pid);	remote get
`put`(Y, pid, X);	remote put
`flush`(pid)	flush

Semantics. Let *Procs* be a finite set of process identifiers and $p \in Procs$ a process id. Let *Vars* be the set of all variables. We assume that each variable is uniquely identified (no two variables have the same name) and we define $proc : Vars \rightarrow Procs$ as the function mapping each variable to the process where it is declared. We define a transition system as a tuple (s_0, Σ, T), where Σ is the set of program states, $s_0 \in \Sigma$ is the initial state, and $T \subseteq \Sigma \times Actions \times \Sigma$ is a transition relation. The *Actions* set contains all statements in the simple language, and the nic_r and nic_w actions which correspond to the asynchronous

non-deterministic execution of the remote statements:

$$Actions = \{put, get, nic_r, nic_w, load, store, flush\}$$

A program state $s \in \Sigma$ is a tuple $\langle pc, M, R, W\rangle$, where:

- $pc : Procs \to Labels$ is the map from process identifiers to labels. The next label that comes after label $l \in Labels$ is denoted by $n(l)$.
- $M : Vars \to D$ is the state of the memory, mapping each variable to a value in the domain D.
- R is a mapping from process ids to the set of pending remote read operations triggered by the given process. For $p \in Procs$, each read operation $r \in R(p)$ has a variable to be read, denoted with $var(r) \in Vars$. Additionally, r is mapped to a following write action denoted $succ(r)$.
- W is the mapping from process identifiers to the set of pending remote write operations triggered by the given process. Given $p \in Procs$, for each $w \in W(p)$, we define the variable to be written, $var(w) \in Vars$ and the value to be written $val(w) \in D$.

The initial state s_0 has both, the set of pending reads and the set of pending writes initialized to be empty ($\forall p \in Procs : R(p) = W(p) = \emptyset$). These semantics are introduced by [17] and match the configuration without in-order routing, where operations are not ordered between the same source and destination processes. Each rule corresponds to a transition $s \xrightarrow{a} s'$, where $a \in Actions$ and $s, s' \in \Sigma$, $s = \langle pc, M, R, W\rangle$ and $s' = \langle pc', M', R', W'\rangle$.

When a *put* action is executed, a pending read operation r is added to R, and a following write operation w is declared. The variables read and written by r and w correspond to the arguments of the `put` statement. The *get* action has similar semantics. The execution of the pending read operations in R is non-deterministic and, after reading the value of the target variable, the following write operation is added to the set of pending writes W. Similarly, the pending writes are executed non-deterministically. The local *store* and *load* actions are executed synchronously and interact directly with the memory M (storing a value, or reading from memory), without using the pending operation sets R and W. Assuming process $p \in Procs$ issues a *flush* action, after its execution the set $W(p)$ does not contain any pending write operations to the target process of the flush. Similarly, the set $R(p)$ does not contain any pending read operations from the target process of the flush, and additionally none of the successor write operations of the pending read operations in $R(p)$ write to the target process. Next, we briefly recap standard predicate abstraction, assuming SC.

2.2 Predicate Abstraction under Sequential Consistency

This section illustrates the standard predicate abstraction procedure applied to the example program in Figure 2, assuming SC.

SC semantics. When restricting an RMA program to SC, we assume that all remote operations (e.g., the `get` statements in Figure 2) are executed synchronously. For example, the R = `get`(Y, 1) statement has the same semantics as a normal assignment R = Y. For our example, the predicates sufficient to verify the specification are:

$$V = \{(r \neq 3), (R \neq 3), (Y \neq 3), (X \neq 3)\}$$

This set of predicates can be determined either manually or using a counter example guided refinement loop. The result of applying predicate abstraction on the example program using the predicates in V is a concurrent boolean program that soundly represents all possible behaviors of the original program.

Boolean program construction. The resulting boolean program has four boolean variables $\{B_1, B_2, B_3, B_4\}$ (one for each predicate in V). For each statement of the program P, standard predicate abstraction computes how the statement changes the values of the predicates. For instance, statement R = `get`(Y, 1) (which for SC we interpret as R = Y) at line 5 in `Process` 2 assigns to variable B_2 (corresponding to the predicate $(R \neq 3)$) the value of B_3. We say that the predicate B_2 is updated using the *cube* of size 1 containing the predicate B_3. Intuitively, $(R \neq 3)$ holds after the statement R = `get`(Y, 1) if $(Y \neq 3)$ holds before the statement. More details about standard predicate abstraction are presented in subsection 4.1.

The complexity of building the boolean program using standard predicate abstraction is exponential in the number of predicates in V ([11, 28]) and its main component is the search of *cubes* (conjunctions of predicates or negated predicates that imply a given formula). Optimizations such as bounding the size of cubes to a constant k lead to a complexity of $|V|^k$, by building a coarser abstraction, therefore losing precision.

2.3 Predicate Abstraction for RMA Programs

We next illustrate our RMA verification approach which is based on *extrapolating* the proof of the program under sequential consistency (discussed in more detail in subsection 2.2).

Step 1: Verify P under SC. The input for this step is the program P (Figure 2) and the set of predicates V shown in subsection 2.2. Here, we assume all remote statements are executed *synchronously.* After the program is verified, the resulting boolean program $B = \mathcal{BP}(P, V)$ will be used for extrapolation in the third step of our approach.

Step 2: Construct the reduced program $\overline{P}$. We reduce the problem of verifying P under RMA semantics to the problem of verifying a new program $\overline{P}$ under SC. The program $\overline{P}$ non-deterministically accounts for all possible asynchronous behaviors of P under RMA.

Auxiliary variables. To construct $\overline{P}$, we introduce auxiliary variables of two types: sets and boolean flags. Additionally, we use two methods: addToSet, that takes as arguments a set and an element, and adds this element to the set, and randomElem, that takes as input a set and returns a random element of the set. The sets accumulate all values that can be read by a `get` statement, or all possible values that can be written remotely by a `put` statement. In our example, we introduce two set variables: X1Set and Y1Set. For example, variable X1Set stores all values of variable X that the `put` statement at line 5 in Figure 2 can write to Y. Initially, X1Set and Y1Set are empty. A boolean flag is introduced for each remote statement. It represents whether the remote statement is pending to be executed asynchronously by the underlying network. For the example in Figure 2, we introduce variables Put1Active and Get1Active, initially set to `false`, corresponding to the `put` and `get` statements.

Statement translation. Next, for each statement of P, we generate a corresponding code in $\overline{P}$. The result of translating the program in Figure 2 is partly shown in Figure 3 (only translation for lines $1-4$ of the original program is shown).

```
    Process 2:
1   shared R = 0, X = 2,
2   Put1Active = false,
3   Get1Active = false,
4   Y1Set = ∅,
5   X1Set = ∅;
6   local r;
7   // put(Y, 1, X);
8   if( !Put1Active )
9     Put1Active = true;
10    X1Set = {X}
11  else
12    addToSet(X1Set,X);
13  // X=3;
14  if(Put1Active && ⋆)
15    Y = randomElem(X1Set);
16    addToSet(Y1Set,Y);
17    if(⋆) Put1Active=false;
18  store X = 3;
19  if( Put1Active )
20      addToSet(X1Set,X);
21  ⋮
```

Fig. 3. Running example translation excerpt of $\overline{P}$: this program contains the RMA behaviors affecting the original program P.

Lines $2-5$ initialize the introduced auxiliary variables (initializing boolean flags such as Put1Active and sets such as X1Set). Lines $7-12$ represent the translation of the `put` statement, where the flag Put1Active is set to `true` and the current value of variable X is added to X1Set. Next, the $X = 3$ statement of program P corresponds to lines 13-20. If the `put` operation is active, then the value 3 is added to X1Set (in line 20). Before it, remote operations are non-deterministically executed. Lines 14-17 represent the non-deterministic asynchronous execution of a pending remote operation. In this case, the only pending operation corresponds to the `put` statement. The $\star$ in the condition of line 14 represents a non-deterministic choice of whether to take the branch or not. Line 15 selects a random element from the set X1Set and assigns it to Y. The transformation process is described in detail in subsection 3.1.

Predicate extrapolation. For the newly generated program $\overline{P}$ shown in Figure 3, our technique automatically extrapolates the set of predicates V to a new set of

predicates $\overline{V} = \mathcal{EP}(V)$. Given the newly introduced set variables, we generate predicates that are universally quantified over all the elements of a set. For example, given the initial SC predicate $(X \neq 3)$, we generate the quantified predicate $\forall e \in X1Set : e \neq 3$. For simplicity, we denote the predicate as $(X1Set \neq 3)$. We assign a special logic to the case where $(X1Set \neq 3)$ is false - it implies that all the elements in *X1Set* are equal to 3:

$$(X1Set \neq 3) = \begin{cases} true, \ \forall e \in X1Set : e \neq 3 \\ false, \forall e \in X1Set : e = 3 \\ \star, \quad\quad otherwise \end{cases}$$

This predicate allows to keep track of the values added to the set. When all the elements of the set are different than 3, then the predicate is *true*. *Importantly*, the predicate is *false* only when all the elements of the set are equal to 3; otherwise, the value of the predicate is unknown, and we denote this value by $\star$. The result of applying predicate extrapolation will return the following set:

$$\overline{V} = V \cup \{(X1Set \neq 3), (Put1Active = true)\}$$

For each boolean flag that the program translation introduces (e.g., Put1Active), the predicate extrapolation will add a predicate that tracks the value of the flag. In our example, the new predicate is $(Put1Active = true)$. More details about predicate extrapolation are presented in subsection 4.2.

Step 3: Construct the Extrapolated Boolean Program After performing the second step, we obtain a program $\overline{P}$ and a set of predicates $\overline{V}$. Applying standard predicate abstraction and building a boolean program $\mathcal{BP}(\overline{P}, \overline{V})$ requires a significantly higher computational effort than building $\mathcal{BP}(P, V)$. The reason is that $\overline{P}$ contains more instructions than P that have to be analyzed, and $\overline{V}$ has more predicates than V. Instead, we generalize the idea of predicate extrapolation [20] to boolean program extrapolation: starting from the boolean program constructed for the SC semantics $B = \mathcal{BP}(P, V)$, we construct a new boolean program $\overline{B} = \mathcal{EBP}(B, \overline{P}, \overline{V})$. By extrapolating the boolean program, we avoid performing additional cube search (we do not require calls to an SMT solver) to construct $\overline{B}$, because we extrapolate cubes already found for the construction of B.

For instance, at line 15 of Figure 3, the statement Y = randomElem(X1Set), where a random element of X1Set is selected and assigned to Y, the boolean program extrapolation will use as input the transformers in the boolean program B that correspond to the statement `put`(Y, 1, X) from Figure 2. For the statement `put`(Y, 1, X), the predicate $(Y \neq 3)$ is assigned the value of $(X \neq 3)$. We extrapolate this boolean assignment for the statement `put`(Y, 1, X), and the predicate $(Y \neq 3)$ is assigned the value of the predicate $(X1Set \neq 3)$. If the predicate $(X1Set \neq 3)$ is *true*, then all the elements inside the set X1Set are different than 3. Therefore, selecting a random element of the set and assigning it to X makes the value of $(Y \neq 3)$ *true*. If the predicate $(X1Set \neq 3)$ is *false*, it means that the elements in X1Set are equal to 3, and assigning any of them to Y

makes the predicate $(Y \neq 3)$ *false*. A formal description of the boolean program extrapolation is presented in subsection 4.3.

Step 4: Model Checking We verify if $\overline{B}$, constructed in Step 3, satisfies the specification S, using a model checker. If there is no reachable state in the program that contradicts S, then the verification succeeds. If an error state is discovered, there are two possibilities: either the error state is spurious, and we refine the abstraction, or it is a valid error state, and the original program P needs more `flush` statements such that property S holds under RMA. If we add `flush` in all possible locations the program is guaranteed to verify, since the program is then limited to SC executions.

Verification results. The specification holds under RMA semantics with a single `flush` statement added to the program, after the `put`(Y, 1, X) instruction of line 3. It is important to observe that this allows the program to retain RMA specific behaviours that do not violate the assertion and are not possible under SC. For example, the value $r = 0$ is not possible under SC, while under RMA, the value of r at the end of the program can be 0. This is because an asynchronous `get` operation, can be executed even after `Process` 2 assigns the value of R to r. Although the `get` reads into R the value 2 from Y (that value reached there with the `put` of line 3 and the following `flush`), the line 6 assignment will write the value of R from initialization into r. Adding a `flush` after the `get` statement would eliminate this state under RMA. However, the state satisfies the specification of interest (r $\neq$ 3), and our procedure successfully identifies which `flush` statement is not required.

3 Reduction of RMA programs

In this section, we describe the source-to-source transformation of a program P running under RMA semantics to a new program $\overline{P}$ running under SC semantics, such that $\overline{P}$ soundly approximates P. The main idea is to generate a program $\overline{P}$ which encodes the RMA semantics of program P.

3.1 Reduction: RMA to SC

We define the translation function that takes as input a statement from program P and returns a list of statements of program $\overline{P}$:

$$[\![]\!] : \mathit{Stmt} \rightarrow \mathit{List}\langle \mathit{Stmt} \rangle$$

Set variables. The newly generated program $\overline{P}$ contains, in addition to the variables of program P, two types of auxiliary variables that contribute to capturing the semantics of RMA programs. Let *sid* be a mapping that takes as input a

Table 2. The source to source translation of statements from program P running on RMA to a new program $\overline{P}$ running on sequential consistency. p is the identifier of the process that executes the statement.

$[\![$`store` X = a$]\!]$	$[\![$`put`(Y, pid, X)$]\!]$

remoteOps(X)
`store` X = a;
$\triangleright\ \forall s \in read(X)$:
    `if` $active(s)$
        addToSet($set(s)$,X);
	`if` (! $active(s)$)
	    $active(s)$ = `true`;
	    $set(s)$ = {X};
	`else` addToSet($set(s)$, X);

$[\![$`flush`(pid)$]\!]$	remoteOps(X)

`while`($\bigvee_{s \in remote(p,pid)} active(s)$)
$\triangleright\ \forall s \in chain(remote(p, pid))$:
    `if` ($active(s)$ ∧ ⋆)
      $trg(s)$ = randomElem($set(s)$);
      `if` (⋆) $active(s)$ = `false`;
	`while`(⋆)
	$\triangleright\ \forall s \in chain(write(X))$:
	    `if` ($active(s)$ ∧ ⋆)
	      $trg(s)$ = randomElem($set(s)$);
	      addToSet($Sets(trg(s))$, $trg(s)$);
	      `if` (⋆) $active(s)$ = `false`;

remote statement (`put` or `get`) of program P and returns a unique identifier of that statement. The first type of auxiliary variables are sets. For each remote `get` statement s of the form var_dst = `get`(var_src, pid) we introduce a set variable. The name of the set variable is the concatenation of the var_src variable, the unique identifier of the `get` statement $sid(s)$ and the string "Set". For example, assuming the statement X = `get`(Y, pid) with unique identifier 1, we generate the set variable Y1Set. Similarly, for each `put` statement `put`(var_dst, pid, var_src), we generate a set variable corresponding to var_src and the statement identifier. All the set variables are initially empty. Given a remote statement s, let $set(s)$ be the set variable associated to s in our translation. Let $Sets$ be the set of all set variables of the program and $Sets(X)$ the sets corresponding to a variable X.

Boolean flags. The second type of auxiliary variable is a boolean flag. For each remote `put` and `get` statement, we introduce a boolean variable, with a name that is the concatenation of "Get"/"Put", the statement id $sid(s)$ and the string "Active". For instance, given a statement `put`(Y, pid, X) with statement id equal to 3, we add the boolean flag Put3Active. All the auxiliary boolean variables are initially `false`. Let $Flags$ be the set of all boolean flags. The mapping $active : Sets \rightarrow Flags$ returns the boolean variable that corresponds to the same remote statement as the set variable given as argument.

Chains of remote statements. Given two remote statements s_1 and s_2, the relation $s_1 \circ s_2$ holds if s_1 writes to a variable X and s_2 reads or writes X. Let $\circ^+$ be the transitive closure of $\circ$. We define $chain(s_2)$ as the set of remote statements s

such that $s \circ^+ s_2$. We overload the *chain* operator to sets of remote statements: given S a set of remote statements, $chain(S) = \bigcup_{s \in S} chain(s)$.

Notation. For a given shared variable X, there are potentially several set variables, one for each remote statement that reads the value of X. Let $write(X)$ be the set of remote statements that write to X and $read(X)$ the set of remote statements reading from X. We denote by $remote(p, pid)$ all the statements executed by process p that remotely read or write from process pid. The variable that is written by a remote statement s is $trg(s)$. These functions are used in the source to source translation.

Translation of program statements. Table 2 illustrates the source-to-source translation of program P running on RMA to a new program $\overline{P}$ that runs on SC and captures all the behaviors of P.

Non-deterministic execution of remote operations. Since RMA remote operations are executed asynchronously, they could be executed at any point in the program after the statement that triggers them. An exact translation would non-deterministically execute, in any order, every pending RMA statement at each program point. However, the state space of the resulting program would grow significantly, making the verification more challenging.

In our approach, the resulting program $\overline{P}$ contains, at specific points, code that executes the pending remote operations non-deterministically. This code is described in Table 2 as remoteOps, and is parameterized by a shared variable X. The statements executed non-deterministically are all the remote statements that write to X and all the remote statements that form *chains* with the statements writing to X, denoted $chain(write(X))$. For each statement $s \in chain(write(X))$, if the active flag corresponding to s ($active(s)$) is `true`, then non-deterministically ($\star$) assign to the target variable of s ($trg(s)$) a random element of the set variable corresponding to s ($set(s)$). Next, the active flag is non-deterministically set to `false` (there can be potentially several instances of statement s pending, in case s is executed in a loop).

Local store. The statement `store` X = a writes a local variable a or numerical value to a shared variable X that belongs to the process executing the statement. In the translation, we first add the remoteOps(X) code. Next, we add the value of a to the corresponding sets of the remote statements to all the set variables that correspond to remote statements reading from X (denoted $read(X)$).

Remote Put. A `put`(Y, pid, X) statement reads a shared variable X of the local process and writes its value to the shared variable Y at the remote process pid. This operation is done asynchronously by the underlying network. There are potentially several values that the local read from X can observe, when X is modified by the program, following the remote `put` statement. Similarly, the remote write operation to Y at the remote process pid happens non-deterministically after the read from X.

For our reduction, the statement s = `put`(Y, pid, X) is translated by first checking if the flag variable ($active(s)$) is `false`. In this case, the s becomes active (by setting $active(s)$ =`true`), and we initialize $set(s)$ with the current value of X ($set(s) = \{X\}$). If the $active(s)$ flag is `true` (that can be the case if s is part of a loop), then the current value of X is added to $set(s)$.

Flush. The `flush`(pid) statement makes sure that after its execution all active RMA operations from the current process (p) to the remote process pid (denoted by $remote(p, pid)$) are executed. We translate the `flush` statement as a loop that executes pending operations as long as at least one of the RMA statements in $remote(p, pid)$ is still pending. The loop contains non-deterministic statements to execute each of the pending statements in $remote(p, pid)$ and statements that create *chains* with the pending statements in $remote(p, pid)$. This translation is sound, as it covers all the possible orders of execution of the pending statements.

4 Predicate Abstraction for RMA Programs

This section describes how to adapt predicate abstraction to the task of verifying RMA programs. The key idea is to cheaply build an RMA proof from the SC proof by reusing predicates and transformers of the SC program proof. Note that the safety property that we want to prove remains the same as in the SC case. We begin by describing standard construction of predicate abstraction for SC.

4.1 Predicate Abstraction

Predicate abstraction [11, 28] is a form of abstract interpretation that employs Cartesian abstraction over a set of predicates. Given a program P, and vocabulary (set of predicates) $V = \{p_1, \dots, p_n\}$ with corresponding boolean variables $\hat{V} = \{b_1, \dots, b_n\}$, predicate abstraction constructs a boolean program $\mathcal{BP}(P, V)$ that conservatively represents behaviors of P using only boolean variables from $\hat{V}$ (corresponding to predicates in V). We use $[p_i]$ to denote the boolean variable b_i corresponding to p_i. We note that the mapping is a bijection. We similarly extend $[\cdot]$ to any boolean function φ.

Constructing $\mathcal{BP}(P, V)$. A literal is a predicate $p \in V$ or its negation. A *cube* is a conjunction of literals, and the size of a cube is the number of literals it contains. The concrete (symbolic) domain is defined as formulae over all possible predicates. The abstract domain contains all the cubes over the variables $p_i \in V$. Predicate strengthening F_V maps a formula φ from the concrete domain to the largest disjunction of cubes (over V), d, such that $d \Longrightarrow \varphi$. The abstract transformer of a statement st, w.r.t. a given vocabulary V, can be computed using the weakest-precondition ([11]), while performing implication checks using an SMT solver. For each $b_i \in \hat{V}$ the abstract transformer generates:

$$b_i = choose([F_V(wp(st, p_i))], [F_V(wp(st, \neg p_i,))])$$

where $choose(\varphi_t, \varphi_f) = \begin{cases} true, & \varphi_t \text{ evaluates to } true \\ false, & \text{only } \varphi_f \text{ evaluates to } true \\ \star, & \text{otherwise} \end{cases}$

For example, given the predicates $V = \{(X > 0), (Y > 0), (Z > 2)\}$, the corresponding boolean variables $\hat{V} = \{b_1, b_2, b_3\}$, and a statement X = Y + Z, predicate abstraction generates the abstract transformer: $b_1 = choose(b_2 \wedge b_3, false)$. After executing the statement X = Y + Z, $(X > 0)$ holds if $(Y > 0)$ and $(Z > 2)$ hold before the statement, otherwise $(X > 0)$ becomes $\star$. Different predicate abstraction techniques use different heuristics for reducing the number of calls to the prover.

4.2 Predicate Extrapolation, $\overline{V} = \mathcal{EP}(V)$

After the SC predicate abstraction is successfully completed, we *extrapolate* the set V of SC predicates, and we obtain $\overline{V}$, the predicates for the reduced program $\overline{P}$. The set $\overline{V}$ of predicates consists of: (i) the predicates V from the SC proof, (ii) universally quantified predicates for each set variable, based on the extrapolation of SC predicates, and (iii) predicates for the boolean flag variables.

Extrapolation of SC predicates for each set variable. A sound optimization of the source-to-source translation is to generate just one variable *XSet* for each shared variable X, instead of one set variable per remote statement. In the rest of the paper, we denote *XSet* the set corresponding to X. The abstraction accounts for the set variables and tracks predicates that hold for the values contained in these sets. We generate for each predicate $p \in V$, that references a shared variable X, a corresponding predicate for *XSet*, $\forall e \in XSet : p[^e/_X]$. The newly generated predicate contains a universal quantifier over all the elements of the set. We denote this predicate as $p[^{XSet}/_X]$.

Logic of the predicates over set variables. A predicate $\forall e \in XSet : p[^e/_X]$ over a set variable *XSet* is *true* if and only if predicate p is *true* for every element of *XSet*. However, we refine the case where the predicate $\forall e \in XSet : p[^e/_X]$ is *false*: if $p[^e/_X]$ is false for every element $e \in XSet$. Overall, the set predicates have the following logic:

$$p[^{XSet}/_X] = \begin{cases} true & \forall e \in XSet : p[^e/_X] \\ false & \forall e \in XSet : \neg p[^e/_X] \\ \star & otherwise \end{cases}$$

For example, assume we are given *XSet* (initially empty), the predicate $p = (X > 5)$, such that $p[^{XSet}/_X] = (XSet > 5)$ and a sequence of addToSet statements that successively add the values 6, 7 and 4 to *XSet*. After executing the statement addToSet(XSet, 6), the predicate $p[^{XSet}/_X]$ becomes *true*. After addToSet(XSet, 7), the predicate $p[^{XSet}/_X]$ remains *true*. After addToSet(XSet, 4), $p[^{XSet}/_X]$ becomes $\star$, because neither all elements in *XSet* are greater than 5 nor it is the case that all elements of *XSet* are less or equal than 5.

If we select a random element from *XSet* using T = randomElem(XSet), then the value of a predicate $p[^{XSet}/_{X}]$ is the same as $p[^{e}/_{X}]$, where $e \in XSet$. This can be used to derive the value of predicates that contain variable T. For example, given the predicates $(T > 3)$ and $(XSet > 5)$, if $(XSet > 5)$ is *true* before T = randomElem(XSet), then $(T > 3)$ is *true* after the statement is executed. Similarly, if $(XSet > 5)$ is *false*, then a predicate such as $(T > 5)$ becomes *false* after the statement, by using the special logic we assign to the set predicates.

In our implementation, for predicates that reference two shared variables (e.g., $(X < Y)$) we extrapolate for each shared variable separately and do not generate a predicate involving two set variables (e.g., $(XSet < Y)$ and $(X < YSet)$ are generated, while $(XSet < YSet)$ is not generated). This provides a good trade-off between precision and efficiency. We observe that this over-approximation is precise enough and there is no need to track directly the relation between two set variables. We show in subsection 4.3 how to soundly handle this abstraction.

Generation of predicates for the boolean flags. For each remote operation (for example a `get` operation at label *lbl*) the translation presented in subsection 3.1 generates a boolean flag variable to indicate when the remote operation (or an instance of the operation, in case it is executed in a loop) is pending to be executed (for the `get` operation, a get_lbl_flag boolean variable). For each such boolean variable we generate a corresponding predicate that captures the boolean flag state (e.g., for the `get` operation at label *lbl*, we generate the predicate $(get_lbl_flag = 1)$).

4.3 Boolean Program Extrapolation, $\overline{B} = \mathcal{EBP}(\overline{P}, \overline{V}, B)$

Given the extrapolated predicates $\overline{V}$, a standard construction of the boolean program is typically quite expensive, because the number of predicates $|\overline{V}|$ and the size of the program $|\overline{P}|$ are significantly larger than in the SC case. This observation was shown in [20] for relaxed memory models, a work which proposed cube extrapolation to reduce the number of calls to an underlying theorem prover. In our work, in addition to handling a more complex memory model (RMA) and generating more complex predicates that contain quantifiers, we introduce a novel boolean program extrapolation method that builds a boolean program without any calls to the theorem prover. The resulting boolean program $\overline{B}$ is a sound over-approximation of $BP(\overline{P}, \overline{V})$.

Transformers for the set operations. The main difference between programs P and $\overline{P}$ is that the latter contains set variables and statements that operate on the set variables (set initialization, addToSet and randomElem). The predicates in $\overline{V}$ refer to the set variables and we next describe how to compute the transformers for the set operators. We again note that attempting to directly calculate their transformers would require a large number of calls to the theorem prover.

Transformers for set initialization. We construct the translation such that all the set initialization statements have the form XSet = {X}). A set XSet is always

initialized with the singleton set containing the value of the variable X (the variable to which the set corresponds). As shown in subsection 4.2, the predicate extrapolation generates for every predicate $p \in V$ that contains X a predicate $p[^{XSet}/_X]$. Therefore, after executing the statement XSet = {X}, the predicates containing XSet have the same value as the predicates containing X:

$$p[^{XSet}/_X] = p$$

Transformers for addToSet. All addToSet have the form addToSet(XSet, X), adding the value of a variable X to the set XSet that corresponds to that variable. The predicates $p[^{XSet}/_X] \in \overline{V}$ are updated after addToSet(XSet, X) such that:

$$p[^{XSet}/_X] = choose(p[^{XSet}/_X] \wedge p, \neg p[^{XSet}/_X] \wedge \neg p)$$

If all the elements in XSet satisfy the predicate p and the value of X satisfies P, then, after adding the value of X to XSet, all the elements of XSet still satisfy p.

Transformers for randomElem. Every statement Y = randomElem(XSet) in program $\overline{P}$ corresponds to a remote operation such as Y = `get`(X) or `put`(Y, X) from program P. For the SC verification, these remote statements are assumed to be synchronous assignments of the form Y = X. During the SC predicate abstraction, for each predicate $p \in V$ that contains Y, we compute φ_t and φ_f, the disjunctions of cubes that appear as arguments of the *choose* function that updates p: $p = choose(\varphi_t, \varphi_f)$. In the case of the Y = randomElem(XSet) statement, for all predicates $p \in V$ that contain Y, we update them using the formula:

$$p = choose(\varphi_t[^{XSet}/_X], \varphi_f[^{XSet}/_X])$$

Consider an example with $(X > 7), (Y > 5) \in V$ and a statement Y = `get`(X) in P with the SC transformer $(Y > 5) = choose((X > 7), false)$. For the corresponding Y = randomElem(XSet) statement in $\overline{P}$, we generate the transformer $(Y > 5) = choose((XSet > 7), false)$.

The extrapolated predicates $p \in \overline{V}$ that contain both Y and at least one set variable are updated to $\star$ after the Y = randomElem(XSet) statement. This sound over-approximation is required because $\overline{V}$ has no predicates that contain more than one set variable. For example, given the predicates $(X \geq Z), (Y \geq X), (Y \geq Z) \in V$, the extrapolated predicate $(Y \geq ZSet) \in \overline{V}$, after a statement Y = randomElem(XSet) the predicate $(Y \geq ZSet)$ is set to $\star$, as we do not track the predicate $(XSet \geq ZSet)$ that is required for a more precise transformer.

5 Experimental Evaluation

We implemented an analysis tool for RMA programs based on the method described so far. In this section, we discuss our experimental evaluation of the tool on a number of challenging concurrent algorithms running on RMA networks. The experiments ran on an Intel(R) Xeon(R) 2.13GHz with 250GB RAM. The

first research question is whether predicate and boolean program extrapolation are sufficiently scalable to verify all benchmarks. The second question deals with the precision of the abstractions we introduced and whether we can compute the smallest required `flush` placement for each program such that our tool is precise enough to prove that the specification holds under RMA.

Benchmarks. We tested our analyzer on 14 challenging concurrent algorithms: Dekker [23], Peterson [38], Szymansky [42] mutual exclusion algorithms, an Alternating Bit Protocol (ABP), an Array-based Lock-Free Queue, Lamport's Bakery algorithm [34], the Ticket locking algorithm [8], the Pc1, Pgsql, Kessel, BlueTooth, Sober, Driver Qw, loop2_TLM programs as defined in [14], and an RMA Lock [41]. The benchmarks have two or three processes and the number of lines of code is between 25 and 85. Several programs have an infinite number of states (ABP, Queue, Bakery, Ticket). The safety properties are either mutual exclusion or reachability invariants involving labels of different processes. For each benchmark, the safety property is the same for both SC and RMA.

5.1 Prototype Implementation

We implemented the RMA analyzer in Java (around 9,000 lines of code). For the cube search (when building the boolean program for SC verification), the tool uses Z3 [22] as an underlying SMT solver. We use the 3-valued model checker Fender (implemented in Java) to check if the boolean program satisfies the specification. Fender also uses Z3 for abstraction refinement. We made minor changes to the error trace construction and interpolation methods of Fender in order to accommodate the RMA abstraction based on sets.

Flush search. For an input program, we initially add a `flush` statement after each RMA remote statement (`put` or `get`). Alternatively, the user can suggest a different initial `flush` placement. The analyzer starts checking the input program using *all* the flushes of the initial `flush` placement. If the analyzer successfully verifies the program, then the `flush` search process continues by removing one `flush` statement, updating the boolean program and rechecking the property using Fender (no need to rerun the predicate abstraction).

We develop a search procedure for the smallest placement of `flush` statements for which our tool successfully proves that the program satisfies its specification under RMA. We choose a mix between breadth-first and depth-first search. In the first phase (breadth-first search of depth 1), we repeatedly check the program while removing one of the `flush` statements of the initial placement. This phase identifies the `flush` statements that are always needed for the program to satisfy the specification. In the second phase, the tool performs a depth-first search, while trying to remove only `flush` statements that were successfully removed in the first phase. This hybrid solution is much faster than a simple depth-first or breadth-first search, especially for the cases where the number of `flush` statements needed is small. Finally, the search returns one or several solutions of flush placements that make the program satisfy the desired property.

Table 3. Experimental results showing verification of a number of algorithms on both SC and RMA models.

	SC predicate abstraction			RMA predicate abstraction			
Algorithm	$\|V\|$	$\mathcal{BP}(P,V)$ (s)	$B(loc)$	$\|\overline{V}\|$	$\overline{B}$ (loc)	Fender (s)	Min `flush`
Dekker	11	1	498	29	2068	876	4/12
Peterson	10	1	356	21	1045	4	4/7
Abp	16	1	485	20	662	1	1/2
Pc1	18	2	658	35	3797	126	2/7
Pgsql	12	1	418	18	1549	1	2/4
Qw	13	2	487	29	1544	1345	4/5
Sober	23	8	831	48	8466	10	0/9
Kessel	18	3	534	36	1621	16	5/10
Loop2_TLM	29	165	1068	43	1986	3960	4/4
Szymanski	34	228	1182	64	7081	316	7/14
Queue	13	24	572	22	1104	13	1/2
Ticket	17	114	640	43	3615	4320	5/6
Bakery	19	330	828	41	2947	288	6/10
RMA Lock	24	50	763	60	5932	65679	9/18

5.2 Experimental Results

The results of the analysis are presented in Table 3. For the first part of our analysis, we perform the verification of the programs assuming SC.

Meaning of table columns. $|V|$ represents the number of predicates used for SC verification. To obtain these predicates, we started with a manually selected set, then used abstraction refinement to find the sufficient set to verify the program under SC. The $\mathcal{BP}(P,V)$ (s) column records the duration in seconds of building the boolean program abstraction. Most of this time (95%) is spent in the SMT solver, used for the cube search. $B(loc)$ shows the number of lines of code in the resulting boolean program. Checking the SC boolean program with Fender takes for each algorithm a small number of seconds, therefore we omit it from the results table. In the second step of our analysis, we perform the boolean program extrapolation, based on the SC boolean program B. $|\overline{V}|$ is the number of predicates after extrapolating the SC predicates V ($\overline{V} = \mathcal{EP}(V)$). The column $\overline{B}(loc)$ shows the number of lines of code of the extrapolated boolean program $\overline{B}$. The *Fender (s)* column shows the runtime of Fender for checking whether $\overline{B}$ satisfies the specification. Finally, *Min `flush`* is the result of the search for the minimal number of `flush` statements required for the program to satisfy its specification under RMA. The first number is the smallest number of flushes for which the verification succeeds, and the second number is the number of flushes of the initial `flush` placement.

On average, the extrapolated boolean program is 5 times larger than the SC boolean program. The resulting extrapolated predicates are twice as many, on average, compared to the original predicates $|V|$ (note that $V \subset \overline{V}$). We obtain

larger running times for the Ticket and Loop2_TLM benchmarks, due to the high number of predicates and the complexity of the programs.

Scalability of Boolean program extrapolation. The boolean program extrapolation that constructs $\overline{B}$ takes under a second for each benchmark. If we took the approach of using standard predicate abstraction of the reduced program $\overline{P}$, the time would be significantly higher. For instance, we experimented with the Bakery mutual exclusion algorithm, and the standard approach took over three hours (compared to sub-second times for the extrapolation). The precision of the two boolean programs is similar, as the same minimal `flush` placements are found for both. This shows the advantage of our extrapolation method.

Extrapolation precision and minimal `flush` placement. The extrapolated boolean program is precise enough to remove `flush` statements and verify the property for all benchmark algorithms (except Loop2_TLM). For Loop2_TLM, the model checker timed out after two hours, while checking the boolean program with a `flush` removed. Surprisingly, the Sober algorithm does not require any `flush` statement under RMA. This is due to the algorithm already executing the remote operations in loops that have the same effect as a `flush` (by checking in their condition the value returned by the `get` statement). Comparing these `flush` placements with other memory models (x86 TSO, PSO), is challenging, due to the one sided aspect of the remote operations. Two `store` operations to a shared variable X, one in each thread, under TSO, become a `store` and a `put` in the RMA program, since X belongs to on one of the processes.

6 Related Work

Remote Memory Access (RMA) Programming. The semantics of MPI-3 RMA have been first described, informally, by [31]. The work of [17] introduces operational semantics for Partitioned Global Address Spaces (PGAS), which follow the same principles as RMA programs. The focus in their work is analysing robustness of the programs using PGAS. Our work, on the other hand, focuses on proving that safety specifications hold for a program under RMA by using predicate abstraction. Axiomatic semantics of the core functionality of RMA programs are introduced in [19], which shows the benefits of formal specifications in discovering inconsistencies in existing RMA libraries.

Program Analysis under Weak Memory Models There exists significant body of work in automatically verifying programs and synthesizing fences required for the correctness of programs running under relaxed memory models such as x86 TSO, PSO, Power, C11. This is the first work that verifies infinite-state concurrent programs running on RMA. The work closest to ours is [20], that introduces predicate extrapolation and cube extrapolation for verifying programs under PSO and x86 TSO (more restricted than RMA). While cube extrapolation reduces the search space of cubes when constructing the boolean program, in this

work we introduce complete boolean program extrapolation that side-steps cube search while building the abstraction. Another important difference is that, while [20] abstracts only bounded store buffers, in this work we handle unbounded sets of pending operations via sets and quantified predicates. This results in potentially less `flush` operations needed to enforce the specification. The work of [3] defines a general framework for verifying programs under weak memory models, based on the axiomatic semantics. In our work, we rely on operational semantics of RMA for the source-to-source reduction to SC. The work of [2] uses predicate abstraction to verify x86 TSO programs, while discovering predicates using traditional refinement techniques. In our work, based on a more strict semantics (SC), we directly discover the abstraction for weaker semantics (RMA) via extrapolation. Work on predicate abstraction for infinite-state concurrent programs assuming SC and using compositional methods such as Owicki-Gries and rely-guarantee is presented in [29, 30].

Reduction to SC The reduction of verifying programs under weak memory models to verification under SC via program transformation is also used in [5, 10, 21, 36]. This work introduces a new transformation and abstraction for RMA programs, that is precise enough to verify the program specifications while using a reduced number of `flush` statements. Works by [15, 16, 33, 35] consider verification of finite-state programs under weak memory models, considering just some of the sources of infinite-state programs (e.g. unbounded store buffers or infinite variable domains). Infinite-state programs are handled in [4] for x86 TSO. In recent work, [1] explores the advantages of alternative semantics for TSO (replacing store buffers with load buffers) that is more efficiently verified. In the reduction step of our work, the auxiliary set variables resemble load buffers, because when a remote write operation is performed, the value to be written is selected randomly from the set, which collects all values that the corresponding remote read operation might have. [39] introduces a procedure that detects unexpected executions that might occur when porting the program from a source to a target memory consistency model.

7 Conclusion

We introduced the first automatic verification technique for programs running on RMA networks. The key idea is abstraction *extrapolation*: automatically build an abstraction of the program for a relaxed memory model such as RMA, based on an existing abstraction of the program under SC. We implemented the predicate and boolean extrapolation methods and we successfully verified several challenging concurrent algorithms running on RMA. To our knowledge, this the first time these programs have been verified on the RMA memory model. We believe this work takes a step towards applying proof extrapolation techniques to other hardware or software relaxed memory consistency models.

References

1. Abdulla, P. A., Atig, M. F., Bouajjani, A., and Ngo, T. P. The benefits of duality in verifying concurrent programs under TSO. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada* (2016), J. Desharnais and R. Jagadeesan, Eds., vol. 59 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 5:1–5:15.
2. Abdulla, P. A., Atig, M. F., Chen, Y.-F., Leonardsson, C., and Rezine, A. Automatic fence insertion in integer programs via predicate abstraction. In *Proceedings of the 19th International Conference on Static Analysis* (Berlin, Heidelberg, 2012), SAS'12, Springer-Verlag, pp. 164–180.
3. Alglave, J., and Cousot, P. Ogre and pythia: an invariance proof method for weak consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 3–18.
4. Alglave, J., Kroening, D., Nimal, V., and Poetzl, D. Don't sit on the fence - A static analysis approach to automatic fence insertion. 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* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 508–524.
5. Alglave, J., Kroening, D., Nimal, V., and Tautschnig, M. *Software Verification for Weak Memory via Program Transformation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 512–532.
6. Allen, F., Almasi, G., Andreoni, W., Beece, D., Berne, B. J., Bright, A., Brunheroto, J., Cascaval, C., Castanos, J., Coteus, P., Crumley, P., Curioni, A., Denneau, M., Donath, W., Eleftheriou, M., Fitch, B., Fleischer, B., Georgiou, C. J., Germain, R., Giampapa, M., Gresh, D., Gupta, M., Haring, R., Ho, H., Hochschild, P., Hummel, S., Jonas, T., Lieber, D., Martyna, G., Maturu, K., Moreira, J., Newns, D., Newton, M., Philhower, R., Picunko, T., Pitera, J., Pitman, M., Rand, R., Royyuru, A., Salapura, V., Sanomiya, A., Shah, R., Sham, Y., Singh, S., Snir, M., Suits, F., Swetz, R., Swope, W. C., Vishnumurthy, N., Ward, T. J. C., Warren, H., and Zhou, R. Blue Gene: A vision for protein science using a petaflop supercomputer. *IBM Syst. J. 40*, 2 (Feb. 2001), 310–327.
7. Alverson, R., Roweth, D., and Kaplan, L. The Gemini system interconnect. In *Proc. of the IEEE Symposium on High Performance Interconnects (HOTI'10)* (2010), IEEE Computer Society, pp. 83–87.
8. Andrews, G. R. *Concurrent programming - principles and practice.* Benjamin/Cummings, 1991.
9. Arimilli, B., Arimilli, R., Chung, V., Clark, S., Denzel, W., Drerup, B., Hoefler, T., Joyner, J., Lewis, J., Li, J., Ni, N., and Rajamony, R. The PERCS high-performance interconnect. In *Proc. of the IEEE Symposium on High Performance Interconnects (HOTI'10)* (Aug. 2010), IEEE Computer Society, pp. 75–82.
10. Atig, M. F., Bouajjani, A., and Parlato, G. *Getting Rid of Store-Buffers in TSO Analysis.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 99–115.
11. Ball, T., Majumdar, R., Millstein, T. D., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001 (2001), M. Burke and M. L. Soffa, Eds., ACM, pp. 203–213.

12. Barrett, B. W., Brightwell, R. B., Pedretti, K. T. T., Wheeler, K. B., Hemmert, K. S., Riesen, R. E., Underwood, K. D., Maccabe, A. B., and Hudson, T. B. The Portals 4.0 network programming interface. Tech. rep., Sandia National Laboratories, 2012. SAND2012-10087.
13. Beck, M., and Kagan, M. Performance evaluation of the RDMA over Ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proc. of the Workshop on Data Center - Converged and Virtual Ethernet Switching (DC-CaVES'11)* (2011), ITCP, pp. 9–15.
14. Bertrand Jeannet. The ConcurInterproc Analyzer, Sept 2017. available at: `http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi` (Sept. 2017).
15. Bouajjani, A., Derevenetc, E., and Meyer, R. Checking and enforcing robustness against TSO. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings* (2013), M. Felleisen and P. Gardner, Eds., vol. 7792 of *Lecture Notes in Computer Science*, Springer, pp. 533–553.
16. Burckhardt, S., and Musuvathi, M. Effective program verification for relaxed memory models. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings* (2008), A. Gupta and S. Malik, Eds., vol. 5123 of *Lecture Notes in Computer Science*, Springer, pp. 107–120.
17. Calin, G., Derevenetc, E., Majumdar, R., and Meyer, R. A theory of partitioned global address spaces. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India* (2013), A. Seth and N. K. Vishnoi, Eds., vol. 24 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 127–139.
18. Cray Inc. Using the GNI and DMAPP APIs. Ver. S-2446-52, March 2014. available at: `http://docs.cray.com/` (Mar. 2014).
19. Dan, A. M., Lam, P., Hoefler, T., and Vechev, M. T. Modeling and analysis of remote memory access programming. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016* (2016), E. Visser and Y. Smaragdakis, Eds., ACM, pp. 129–144.
20. Dan, A. M., Meshman, Y., Vechev, M. T., and Yahav, E. Predicate abstraction for relaxed memory models. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings* (2013), F. Logozzo and M. Fähndrich, Eds., vol. 7935 of *Lecture Notes in Computer Science*, Springer, pp. 84–104.
21. Dan, A. M., Meshman, Y., Vechev, M. T., and Yahav, E. Effective abstractions for verification under relaxed memory models. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings* (2015), D. D'Souza, A. Lal, and K. G. Larsen, Eds., vol. 8931 of *Lecture Notes in Computer Science*, Springer, pp. 449–466.

22. de Moura, L. M., and Bjørner, N. 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* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.
23. Dijkstra, E. Cooperating sequential processes, TR EWD-123. Tech. rep., Technological University, Eindhoven, 1965.
24. Donaldson, A. F., Kaiser, A., Kroening, D., and Wahl, T. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In Gopalakrishnan and Qadeer [27], pp. 356–371.
25. Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., and Reinhard, J. Cray Cascade: A scalable HPC system based on a Dragonfly network. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)* (2012), IEEE Computer Society, pp. 103:1–103:9.
26. Gerstenberger, R., Besta, M., and Hoefler, T. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. of the ACM/IEEE Supercomputing* (2013), SC '13, pp. 53:1–53:12.
27. Gopalakrishnan, G., and Qadeer, S., Eds. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (2011), vol. 6806 of *Lecture Notes in Computer Science*, Springer.
28. Graf, S., and Saïdi, H. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings* (1997), O. Grumberg, Ed., vol. 1254 of *Lecture Notes in Computer Science*, Springer, pp. 72–83.
29. Gupta, A., Popeea, C., and Rybalchenko, A. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 331–344.
30. Gupta, A., Popeea, C., and Rybalchenko, A. Threader: A constraint-based verifier for multi-threaded programs. In Gopalakrishnan and Qadeer [27], pp. 412–417.
31. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W., and Underwood, K. Remote Memory Access Programming in MPI-3. *ACM Transactions on Parallel Computing (TOPC)* (Jan. 2015).
32. Islam, N. S., Rahman, M. W., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C., and Panda, D. K. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 35:1–35:35.
33. Kuperstein, M., Vechev, M. T., and Yahav, E. Partial-coherence abstractions for relaxed memory models. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), M. W. Hall and D. A. Padua, Eds., ACM, pp. 187–198.
34. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (1974), 453–455.

35. Linden, A., and Wolper, P. An automata-based symbolic approach for verifying programs on relaxed memory models. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings* (2010), J. van de Pol and M. Weber, Eds., vol. 6349 of *Lecture Notes in Computer Science*, Springer, pp. 212–226.
36. Meshman, Y., Dan, A. M., Vechev, M. T., and Yahav, E. Synthesis of memory fences via refinement propagation. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings* (2014), M. Müller-Olm and H. Seidl, Eds., vol. 8723 of *Lecture Notes in Computer Science*, Springer, pp. 237–252.
37. OpenFabrics Alliance (OFA). OpenFabrics Enterprise Distribution (OFED) www.openfabrics.org, 2014.
38. Peterson, G. L. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12*, 3 (1981), 115–116.
39. Ponce de León, H., Furbach, F., Heljanko, K., and Meyer, R. Portability analysis for axiomatic memory models. PORTHOS: one tool for all models. *CoRR abs/1702.06704* (2017).
40. Recio, R., Metzler, B., Culley, P., Hilland, J., and Garcia, D. A Remote Direct Memory Access Protocol Specification. RFC 5040, RFC Editor, October 2007.
41. Schmid, P., Besta, M., and Hoefler, T. High-Performance Distributed RMA Locks. In *Proceedings of the 25th Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)* (Jun. 2016).
42. Szymanski, B. K. A simple solution to lamport's concurrent programming problem with linear wait. In *International Conference on Supercomputing* (1988), pp. 621–626.
43. The InfiniBand Trade Association. *Infiniband Architecture Spec. Vol. 1, Rel. 1.2.* InfiniBand Trade Association, 2004.

Scalable Approximation of Quantitative Information Flow in Programs

Fabrizio Biondi[1], Michael A. Enescu[2], Annelie Heuser[3], Axel Legay[2], Kuldeep S. Meel[4], Jean Quilbeuf[2]

[1] CentraleSupélec, France
fabrizio.biondi@inria.fr
[2] Inria, France
{mihai.enescu,axel.legay,jean.quilbeuf}@inria.fr
[3] CNRS/IRISA, France
annelie.heuser@irisa.fr
[4] National University of Singapore, Singapore
meel@comp.nus.edu.sg

Abstract. Quantitative information flow measurement techniques have been proven to be successful in detecting leakage of confidential information from programs. Modern approaches are based on formal methods, relying on program analysis to produce a SAT formula representing the program's behavior, and model counting to measure the possible information flow. However, while program analysis scales to large codebases like the OpenSSL project, the formulas produced are too complex for analysis with *precise* model counting. In this paper we use the *approximate* model counter ApproxMC2 to quantify information flow. We show that ApproxMC2 is able to provide a large performance increase for a very small loss of precision, allowing the analysis of SAT formulas produced from complex code. We call the resulting technique ApproxFlow and test it on a large set of benchmarks against the state of the art. Finally, we show that ApproxFlow can evaluate the leakage incurred by the Heartbleed OpenSSL bug, contrarily to the state of the art.

1 Introduction

Finding vulnerabilities in programs is fundamental for producing robust programs as well as for guaranteeing user security and data confidentiality. Due to the increasing complexity of software systems, automated techniques must be deployed to assist architects and engineers in verifying the quality of their code. Among these, *quantitative* techniques have been shown to effectively aid in detecting complex vulnerabilities.

Quantitative information flow (QIF) computation [13] is a powerful quantitative technique to detect information leakage directly at code level. QIF leverages information theory to measure the flow of information between different variables of the program. An unexpectedly large flow of information may characterize a potential leakage of information. In practice, this technique relies on the following:

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 71–93, 2018.
https://doi.org/10.1007/978-3-319-73721-8_4

the maximum amount of information that can leak from a function (known as *channel capacity*) is the logarithm of the number of distinct outputs that the function can produce [16].

Recently, QIF computation based on program analysis and model counting has effectively analyzed codebases of tens of thousands of lines of C code [31]. This technique proceeds as follows. A specific fragment of the program (e.g. a function, or the whole program) is modeled as an information-theoretic channel from its input to its output. Program analysis techniques such as symbolic execution or model checking are used to explore the possible executions of the fragment. Program analysis produces a set of constraints that characterizes these executions. Afterwards, a *model counter* is used to determine the number of distinct outputs of the fragment (e.g. the return values of the function, or the outputs of the program). Finally, the base-2 logarithm of the number of possible outputs gives us the channel capacity in bits.

However, even small programs can result in sets of constraints that are difficult to model count. Complex constraints can result, for instance, from complex program constructions such as pointers in C code. As a result, QIF computation is still not able to discover real-world, high-value security vulnerabilities.

In particular, we consider the analysis of the OpenSSL Heartbleed vulnerability [1] to be an achievable target for QIF computation, and aim to analyze vulnerabilities of this complexity. Channel capacity can be used to detect information leakage in cases like Heartbleed. For instance, if the input of a function has a capacity of 6 bits and the output a capacity of 8 bits, then the function has unexpected behavior. Further investigation can determine the origin of the information that is unaccounted for, e.g. restricted memory that the function is not supposed to have access to. The technique has been shown to be able to help detect and confirm bugs in software [26, 24, 27], and to signal to a developer that there may be bugs in a particular part of the software. Indeed, QIF-based techniques, while not foolproof, can use the a large information flow to a particular part of the program as a hint to a developer in order to narrow down where to look for bugs [31].

However, the model counting step of the procedure is very computationally expensive, since it is $\#P$-complete [32]. On the other hand, since channel capacity is computed as the logarithm of the number of outputs, imprecision in the model counting procedure will result only in minor variations of the computed channel capacity. Hence, it is natural to consider using approximate model counting techniques, where the precision of the result is traded for improved efficiency.

This idea has been investigated by Klebanov et al. [22]. However, in Section 5.1 we show that the approach in [22] is fundamentally incorrect, requiring a different technique.

For these reasons, in this paper we propose ApproxFlow, a new QIF computation technique to tackle problems for which precise model counting is not efficient enough. ApproxFlow is based on the ApproxMC2 tool implemented by Chakraborty et al. [12]. We show that ApproxFlow vastly outperforms the state of the art on all but a few of the benchmarks, including on many cases in which

no other tool is able to provide an answer, making it the most efficient tool for QIF computation currently available. The contributions of this paper are:

- We present ApproxFlow, a technique to quantify information flow for deterministic C programs based on the approximate projected model counter ApproxMC2;
- We show that a small decrease in the approximation precision can yield large performance improvements, allowing ApproxFlow to scale to complex cases with minimal reduction in the result's usefulness;
- We show that the technique presented in [22] is incorrect due to some mistakes in its underlying theoretical results;
- We evaluate ApproxMC2 against the precise projected counter sharpCDCL on a large set of benchmarks, showing that the former generally yields orders of magnitude better performance at the cost of a small decrease in precision;
- We use ApproxFlow's improved scalability to model and compute the leakage of the code in the Heartbleed bug [1], unlike previous QIF techniques.

The rest of the paper is structured as follows. Section 2 introduces technical background and notation, and Section 3 discusses related work. We describe our technique ApproxFlow Section 4 and evaluate it in Section 5. Section 6 presents the Heartbleed case study. Section 7 provides additional discussion, and Section 8 concludes the paper.

2 Background

This section introduces the background and notations used in this paper.

Entropy and channel capacity. Let $\mathcal{X}$ be a discrete finite *sample set* and $\rho(\mathcal{X})$ a probability distribution on it, where the probability of an outcome $x \in \mathcal{X}$ is denoted $\Pr[x]$. The *entropy* $H(\rho(\mathcal{X}))$ of a probability distribution $\rho(\mathcal{X})$, measured in bits, is defined as $H(\rho(\mathcal{X})) = -\sum_{x\in\mathcal{X}} \Pr[x] \cdot \log_2 \Pr[x]$. The *conditional entropy* $H(\rho(\mathcal{Y}|\mathcal{X}))$ of the conditional probability distribution $\rho(\mathcal{Y}|\mathcal{X})$, is defined as $H(\rho(\mathcal{Y}|\mathcal{X})) = -\sum_{x\in\mathcal{X}} \Pr[y|x] \cdot \log_2 \Pr[y|x]$, where $\Pr[y|x]$ denotes the probability of an outcome $y \in \mathcal{Y}$ given that an outcome $x \in \mathcal{X}$ has already occurred.

Define a *deterministic channel* D as a triple $(\mathcal{I}, \mathcal{O}, F)$ where $\mathcal{I}$ (inputs) and $\mathcal{O}$ (outputs) are discrete finite sample sets and F is a function $\mathcal{I} \to \mathcal{O}$ defining which output $o \in \mathcal{O}$ is produced for each input $i \in \mathcal{I}$. Hence, any probability distribution $\rho(\mathcal{I})$ on the input $\mathcal{I}$ induces a probability distribution $\rho(\mathcal{O})$ on the output $\mathcal{O}$ via F. The *channel capacity* of D is defined as $C(D) = \max_{\rho(\mathcal{I})} H(\rho(\mathcal{O}))$ where the maximum is taken over all possible probability distributions $\rho(\mathcal{I})$. For a deterministic channel D, it is known [16] that $C(D) = \log_2 |\{o \in \mathcal{O} \text{ s.t. } \Pr[o] > 0\}|$.

The *mutual information* of a deterministic channel D is defined as $I(D) = H(\rho(\mathcal{O})) - H(\rho(\mathcal{O}|\mathcal{I}))$. An alternative but equivalent formulation of D's channel capacity is $C(D) = \max_{i\in\mathcal{I}} I(D)$.

A deterministic program can be regarded as a deterministic channel, where $\mathcal{I}$ and $\mathcal{O}$ represent the possible values for the program's inputs and outputs. In

this case the channel capacity represents the maximum amount of information that can be inferred on the program's inputs by observing its outputs [17].

Model counting. Model counting, or #SAT, is the canonical $\#P$-complete problem, and is the counting analogue of the Boolean satisfiability (SAT) problem [32]. Let ϕ be a *SAT formula* involving variables $\mathcal{V}$, and $a \in \{\texttt{true}, \texttt{false}\}^{\mathcal{V}}$ be a Boolean valuation of $\mathcal{V}$. We say that a is a model of ϕ, denoted $a \vdash \phi$, if ϕ evaluates to `true` when substituting variables with their value in a.

The model count $\#\phi$ of ϕ is the number of valuations that satisfy ϕ:

$$\#\phi = \left|\left\{a \in \{\texttt{true}, \texttt{false}\}^{\mathcal{V}} \mid a \vdash \phi\right\}\right| \ .$$

We introduce the notion of *projection* in the context of model counting [3]. We consider a subset $\mathcal{S} \subseteq \mathcal{V}$ of the variables. Given a Boolean valuation a of $\mathcal{V}$, we naturally define its projection $a|_{\mathcal{S}}$ on $\mathcal{S}$ by restricting the input domain of a to $\mathcal{S}$. The projection $a|_{\mathcal{S}}$ is a Boolean valuation of $\mathcal{S}$.

The projected model count of a SAT formula ϕ on a *projection scope* $\mathcal{S}$ is the number of valuations of $\mathcal{S}$ that can be extended into a model of ϕ:

$$\#\phi_{\mathcal{S}} = \left|\left\{a_{\mathcal{S}} \in \{\texttt{true}, \texttt{false}\}^{\mathcal{S}} \mid \exists a \in \{\texttt{true}, \texttt{false}\}^{\mathcal{V}} a|_{\mathcal{S}} = a_{\mathcal{S}} \wedge a \vdash \phi\right\}\right| \ .$$

Approximate projected model counting with ApproxMC2 Approximate projected model counting [12] refers to the problem of finding an estimate on the projected model count of a SAT formula ϕ onto a subset $\mathcal{S}$ of the variables, as opposed to precise number.

We present the core ideas behind the ApproxMC2 tool used in this paper, and refer the reader to [12, 11] for a full exposition. ApproxMC2 is a *Karp-Luby* (or (ε, δ)) counter [20], which obtains an estimate $\widehat{\#\phi}$ on $\#\phi$ that falls within a factor $1 + \varepsilon$ of $\#\phi$ with a probability of $1 - \delta$, i.e., given a tolerance ε and a probability δ it holds that

$$\Pr\left[(1+\varepsilon)^{-1} \cdot (\#\phi) \leq \widehat{\#\phi} \leq (1+\varepsilon) \cdot (\#\phi)\right] \geq 1 - \delta \ .$$

ApproxMC2 works by randomly partitioning the set of possible models of the SAT formula ϕ projected onto $\mathcal{S} \subseteq \mathcal{V}$ (denoted as $\phi_{\mathcal{S}}$), into roughly equal buckets, performing model counting on this single bucket, and returning this count, multiplied by the number of buckets, as the approximation of the exact projected model count $\#\phi_S$. The partitioning into buckets of roughly equal size is key, and is done using an approach based on *r-wise independent hash functions* [6], adding special randomized *XOR constraints* to the SAT formula. If these randomly-chosen buckets are "too big," the number of buckets is doubled and the procedure is repeated with accordingly smaller buckets.

For the reader's convenience, we present a description of ApproxMC2, the algorithm from [12], in Algorithm 1. We note that the algorithm has a chance to fail to return anything at line 4, when it returns $\perp$. By repeating the algorithm a sufficiently large number of times, we can obtain the desired probability $1-\delta$ that

Algorithm 1: ApproxMC2

```
   Input  : A SAT formula φ with |φ| SAT variables x_1, ..., x_|φ|
   Input  : A projection scope S ⊆ {x_1, ..., x_|φ|}
   Output : An estimate of #φ_S, the number of models of φ projected onto S
1  p ← 1 + 9.84 · (1 + ε/(1+ε)) · (1 + 1/ε)^2                    // pivot value p
2  b ← min(p, #φ_S)              // return p as soon as ≥ p models of #φ_S found
3  if b < p then
4      return b
5  cells ← 2                                                   // Number of cells
6  C ← [ ]                                                         // Empty list
7  for i ← 1 to ⌈17 · log_2(3/δ)⌉ do
8      Choose h at random from H_xor(|S|, |S| − 1)          // Random hash function
9      Choose α at random from {0, 1}^(|S|−1)
10     φ' ← φ ∧ h(S) = α                          // Add random XOR constraint to φ
11     b' ← min(p, #φ'_S)
12     if b' ≥ p then
13         cells ← ⊥
14         models ← ⊥
15     if cells ≠ ⊥ then
16         m ← log_2 LogSATSearch(φ, S, h, α, p, log_2 cells)
17         φ'' ← φ ∧ h^(m)(S) = α^(m)                    // Add XOR constraints to φ
18         models ← min(p, #φ''_S)
19         AppendToList(C, cells · models)
20 return C̃                                                         // Median of C
```

it will succeed. In line 16, the invocation of LogSATSearch refers to a procedure to obtain good values for m. This is beyond the scope of this paper, and we refer to [12] for details. In lines 2 , 11 , and 18, the minimum is computed using a SAT solver which iteratively finds up to p models. Note that this step does not require the usage of a model counter. Thus, the precise model count is not typically computed at these points, unless the formula (augmented with any XOR constraints) has become constrained (small) enough to have p or fewer models.

We emphasize that ApproxMC2 allows us control the tolerance ε. We will show in Section 4.3 how reducing the tolerance can significantly improve the computation time.

3 Related Work

This section presents a short review of work that is related to this paper.

3.1 Quantitative Information Flow

Prior work on QIF has largely followed the paradigm of characterizing the set of a program's outputs. We classify related work into two categories: those which measure channel capacity, and those that measure other kinds of entropy. We make a note that some work in channel capacity formulates their problem in terms of min-entropy, but it is known [25] that for *deterministic* channels, min-entropy and channel capacity are equivalent. In addition, because much

work in QIF considers conditional entropy, we remark that channel capacity corresponds to minimizing the conditional entropy of the output given the input [17]. This is easy to see, as (adopting the definitions and notation from Section 2), $C(D) = \max_{i \in \mathcal{I}} I(D) = \max_{i \in \mathcal{I}}(H(\rho(\mathcal{O})) - H(\rho(\mathcal{O}|\mathcal{I}))$. To maximize $I(D)$, $H(\rho(\mathcal{O}|\mathcal{I}))$ must be minimized.

Channel capacity. Meng and Smith [25] present a method to obtain empirically good upper bounds on the channel capacity of various small synthetic example programs, also contributing to standardizing a set of QIF benchmark programs. In [21], Klebanov et al. show how to obtain precise measurements of the channel capacity (alongside the conditional Shannon entropy) for a number of programs, including the benchmarks from [25], in addition to a number of small synthetic programs and two examples of real C code on the order of magnitude of 100 lines. In [26], Newsome et al. present a compound approach to obtain precise channel capacity measurements for a set of small, synthetic benchmark programs, and very coarse approximations to large, real-world programs up to a million lines. In [31], Val et al. present a way to measure the channel capacity for a number of benchmarks both synthetic and real, showing how to scale to programs up to thousands of lines of code. McCamant and Ernst [24] use a coarse upper-bounding approach for channel capacity based on network flows, showing how to scale to hundreds of thousands of lines of real code and contributing smaller case studies as benchmarks. Phan and Malacaria [27] present a method that is able to analyze and compute upper bounds on the channel capacity for C implementations of several well-known protocols, as well as three few-hundred-line case studies including parts of the Linux kernel. While some of the above work has demonstrated that generating SAT formulas is possible even for large programs, complex program structures such as pointers often result in SAT formulas that are too difficult for model counting. In addition, the various approaches have occupied static points on the precision vs. scale relation, unable to vary precision to obtain significant speedups.

Other QIF measures. In [34], Weigl presents a tool sharpPI, which implements different search heuristics for model counting, applying it to the measurement of Shannon entropy and presenting results for a small, scalable synthetic C program. In [8], Biondi et al. present a technique, implemented in the QUAIL tool [10, 9], to measure Shannon entropy for a number of scalable case studies expressed in a simple imperative language. More recently, Fremont et al. [19] present MaxCount, a novel approximate QIF method effective at finding leaks in programs, with increasing efficacy as the relative size of the leaks increase. In [5], Backes et al. present a technique to analyze small, synthetic programs with respect to various information-theoretic measures. These techniques do not compute the channel capacity, and therefore they are not comparable with our approach. We note that, as a special case of QIF (checking for the existence of a non-zero flow), *qualitative* information flow has been demonstrated to scale to large program sizes, and confirm bugs in real software such as the OpenSSL Heartbleed bug [23]. However, qualitative information flow does not attempt to

measure the *amount* of information flowing through a program, and therefore cannot be directly compared to our work.

Most recently, Biondi et al. present HyLeak [7], a tool based on a combination of channel matrix computation and simulation to compute channel capacity, among other information-theoretic measures.

3.2 Projected Model Counting

Projected model counting is a problem that arises naturally in QIF measurement [26, 21, 31, 27].

In [34], Weigl presents an approach to projected model counting used as part of a QIF measurement technique, implementing several different search heuristics to guide the model counting. In [31], Val et al. present SharpSubSAT, a simple projected model counter as part of a toolchain for measuring channel capacity, which handles projection by removing variables from the formula that are not part of the projection subset. The projected counter SharpCDCL [21] uses a similar technique based on the state-of-the-art model counter sharpSAT. SharpCDCL is in fact the current state-of-the-art tool in projected model counting.

Still, precise model counting often cannot scale to larger problem sizes, prompting the need for approximate methods. Work in approximate model counting has fallen into three categories: counters that provide no theoretical guarantees but empirically yield good estimates on the true count, counters providing a count that represents an upper (or lower) bound on the exact count, and counters that provide an interval within which the exact count falls ((ε, δ)-counters). We are especially interested in these (ε, δ)-counters, for the theoretical guarantees they provide, and for the promise of trading precision for running time. In addition, there are (to the best of our knowledge) no *projected* approximate model counters that fall outside this category.

Klebanov et al. [22] present a counter based on ApproxMC2 [11], with scalability to 10^5 variables and 10^6 clauses. However, as shown in Section 1, this particular counter has some theoretical mistakes, and we cannot consider it among the state of the art. In [12], the authors present a counter based on the one from [11], and demonstrate scalability to formulas with 10^5 variables and 10^6 clauses. Indeed, the counter from [12] is among the state-of-the-art (ε, δ)-counters with projection capabilities.

Recently, Fremont et al. [19] have presented the Maximum Model Counting technique to compute approximate subset model counts, a novel technique based on a partitioning scheme inspired by [12], and using the same underlying algorithm (ApproxMC2) as we do in our technique. In their approach, the effectiveness of the algorithm increases with the number of solutions.

Fig. 1. A high-level view of ApproxFlow's toolchain.

4 Channel Capacity Estimation with ApproxFlow

In this section, we describe ApproxFlow[5], our technique for estimating the channel capacity of a program to a given precision, with the intention of flagging suspicious parts to a developer applying the tool on the program. We restrict ourselves to *deterministic* programs, consistent with our background in Section 2. A high-level view of our approach can be seen in Figure 1. ApproxFlow takes as input a program, and passes it to a model checker to generate a SAT formula representing the program. The SAT formula must be annotated with a projection scope, which is the subset of the variables in the formula that correspond to the original program variables to which we wish to measure channel capacity. We consider these the "output" variables (although they can be SAT variables corresponding to program variables anywhere in the program), while the SAT variables corresponding to the original program inputs are not projected upon or constrained in any way. ApproxFlow then passes this annotated SAT formula to a projection-capable approximate model counter in order to obtain an approximation on the number of models of the formula, projected onto the projection scope. Finally, ApproxFlow takes the logarithm in base 2 to obtain our final measurement – an approximation of the channel capacity of the program. The following subsections provide details on each part of the toolchain.

4.1 Program to SAT Formula

ApproxFlow takes as input a deterministic C program, and uses the model checker CBMC [15] to obtain an annotated SAT formula that represents the original program. The C program may be optionally annotated by the user to specify a given program location or set of program variables to which to measure the channel capacity. In practice, this program annotation is specified using CBMC's assertion facilities. The user can use the assertion `__CPROVER_assert(0,"");` to specify where the formula should be computed. If this annotation is not provided by the user, ApproxFlow automatically converts the program into an equivalent program where all functions in the program have a single return point at the end of each function, and the annotation is automatically placed immediately before this single return point.

[5] ApproxFlow is publicly available at `https://github.com/approxflow/approxflow`.

CBMC performs bounded model checking on the program[6], and outputs a SAT formula in conjunctive normal form (CNF) that represents the constraints on the variables induced by the program. This model checking step is subject to the following limitations: 1) loop unwinding is bounded to a specified depth (always set high enough in our experiments to capture the full behaviour of the program), and 2) the set of possible values for pointers is overapproximated.

For a fuller treatment of the effect of bounded loop unwinding on (precise) channel capacity computation, we refer the reader to [31], but we give a brief treatment of the topic. For some programs, such as server software which includes infinite loops by design, loop unwinding limits the scope of the analysis, and underapproximates program behaviour. Consequently, the channel capacity is also underapproximated. However, it is often the case that an output variable to which we measure leakage already achieves maximum leakage after only a few iterations. In addition, many loops are executed for only a few iterations, and a bound such as 32 (our default) is more than enough to capture the loop's full behaviour. As our goal is *approximate* channel capacity measurement, we argue that our approach is less sensitive than precise approaches to the further approximation induced by bounding the loop unwinding. A similar argument can be made for the *overapproximation* caused by CBMC's conservative pointer analysis, and we again refer the reader to [31] for a discussion in the context of information flow. Both issues are orthogonal to our contributions, as they result directly from CBMC's limitations in performing a more precise analysis; improvements in model checking and formula generation would benefit us directly.

Additionally, CBMC annotates the SAT formula with comments that specify which boolean variables in the SAT formula correspond to the original program variables. In this way, we are able to obtain a SAT formula from CBMC that is annotated with our desired projection scope, which may be then passed to an approximate model counter in order to obtain the number of models of the formula projected onto the specified variables.

4.2 SAT Formula to Channel Capacity

In the second step of our approach, we take as input an annotated SAT formula obtained from the model checker and use a projection-capable approximate model counter to obtain an estimate of the number of models of the projected formula. Specifically, we use an improved implementation of a state-of-the-art approximate model counter ApproxMC2 [12] by Mate Soos and Kuldeep Meel, which is pending publication. For the remainder of the paper, whenever we refer to ApproxMC2, we are actually referring to this improved version.

ApproxMC2 takes a SAT formula in conjunctive normal form (CNF), specified in the DIMACS CNF format [4], with the projection scope specified by special comments in the file. ApproxMC2 provides an approximate number of models of this projected formula within a specified tolerance, with high probability.

[6] We do not discuss model checking in this paper. For a treatment of model checking, please consult [14].

While ApproxMC2 is a state-of-the-art approximate counter, it does have some limitations even when compared to precise tools. ApproxMC2 has significant overhead due to the requirement of adding XOR constraints, which tends to make it perform more poorly in terms of running-time on smaller problems relative to other counters. In addition, ApproxMC2's expected runtime is higher when compared to other counters when it is used to solve formulas that are *dense* in their solution space – that is, formulas which have a large number of models in relation to their formula size (in number of variables). In practice, these limitations are not usually a problem compared to other available counters (precise or approximate). Full details may be found in [12].

Finally, ApproxFlow takes the logarithm in base 2 of this estimated model count in order to obtain an estimation of the channel capacity of the program. Somewhat unique to this problem, it is worth nothing that taking the logarithm of the approximate count exponentially squishes the error in the estimate. In other words, a fairly coarse approximation on the model count can yield good probabilistic bounds on the channel capacity estimate.

4.3 Performance-Precision Trade-off

Using an approximate method naturally leads to a trade-off between precision and performance. Because ApproxMC2 is able to trade a lower precision for a shorter running time, we can choose a trade-off point on the side of shorter running time when a close approximation is not essential (for instance, when an approximate lower bound is the desired outcome, as would be desired when enforcing k-bit policies [26]).

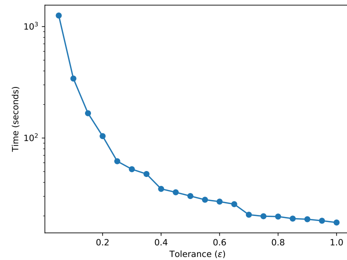


Fig. 2. Precision-time relationship for ApproxMC2. Time (in seconds) is on the vertical axis, and the precision (or *tolerance*) parameter ε is shown on the horizontal axis. Larger values of ε represent more relaxed precision guarantees. All measurements taken were for the preprocessed AppleTalk case (ddp.pp.cnf) from Section 5 with probability $(1 - \delta) = 0.8$.

We evaluate this trade-off for ApproxMC2 on the AppleTalk Linux driver benchmark, ddp.pp (discussed in more detail in Section 5.3). This benchmark exhibits a large enough channel capacity (128 bits) such that a result with a few bits of imprecision is still useful. In addition, it is long-running enough (roughly 20 seconds on our machine, detailed in Section 5) to be largely immune to small variations in time resulting from background CPU usage, making it a good candidate for trading precision for performance.

Figure 2 shows the relationship between precision ε and running time for $0.05 \leq \varepsilon \leq 1$, with a fixed δ of 0.2 (the default value). As time is plotted

on a logarithmic scale, we can see that as soon as we are able to relax the precision/tolerance of the method by increasing epsilon, we gain a dramatic speedup. Since channel capacity is computed as the logarithm of the number of models, the worst case, $\varepsilon = 1$, represents only a single bit of imprecision, yet we observe a speedup factor of 2-3 orders of magnitude over the $\varepsilon = 0.05$ case, which represents an imprecision of $\log_2(1.05) \approx 0.070$ bits. Interestingly, we observed that in all cases, the reported information flow was 128 bits. In practice, the trade-off is even better based on these empirical observations than expected from the theoretical guarantees.

5 Evaluation

In this section, we present an experimental evaluation of ApproxFlow compared to the state-of-the art precise channel capacity measurement tools. We ran all experiments on an Oracle VirtualBox virtual machine with 1 CPU and 8GB of RAM running Linux Mint 18.1 hosted on a Windows 10 machine with a quad-core Intel Core i7 2.9GHz CPU and 16GB of RAM.

5.1 Problems in Klebanov et al. [22]

The usage of approximate model counting to determine channel capacity was previously explored by Klebanov et al. [22]. However, we have been unable to replicate the results in [22]. After further investigation, we have concluded that our inability to replicate such results depends on the fact that the theoretical claims and proofs presented in [22] are incorrect.

The main result of [22] is presented in Theorem 2.12. The theorem aims to show that the algorithm described in the paper terminates with high probability, returning an estimate on the approximate model count. Unfortunately, as result of a mistake in the proof, the probability of termination is overestimated, and the presented algorithm appears to be more effective than it actually is at approximating the true number of solutions. In particular, the proof of this theorem hinges on the following claim (adopting the notation from [22]):

> We now show that there is at least one iteration of the loop (indexed by $m = m'$) such that with a probability of at least $1 - e^{\lfloor -r/2 \rfloor}$ the following is true: the exit condition $c \leq pivot$ holds and the return value $2^{m'} \cdot |\phi_h| \in [(1-\varepsilon)|\phi|, (1+\varepsilon)|\phi|]$

The authors then proceed to prove the above claim and treat it as sufficient for the proof of Theorem 2.12. However, this is incorrect since the above claim is not a sufficient condition for Theorem 2.12. To this end, let us define the event T_i as condition $c \leq pivot$ holds for iteration $m = i$ and the event U_i as $2^i \cdot |\phi_h| \in [(1-\varepsilon)|\phi|, (1+\varepsilon)|\phi|]$. Theorem 2.12 seeks to bound from below the probability of the event S, where $S = \cup_{i=1}^{n}((\cap_{j=1}^{i-1}(\bar{T}_j)) \cap T_i \cap U_i$. Note that, $\Pr[S] \geq \Pr[T_i \cap U_i]$ does not necessarily hold for all i. Therefore, demonstrating

that there exists $m = m'$ such that $\Pr[T_m \wedge U_m] \geq 1 - e^{\lfloor -r/2 \rfloor}$ is not sufficient to support the claim in Theorem 2.12.

In addition, there is an error in the statement of Theorem 2.6. The authors have written the upper bound on the probability as $e^{-\lfloor r/2 \rfloor}$, instead of $e^{\lfloor -r/2 \rfloor}$ [29, Theorem 5]. The authors conclude that the value for *pivot* reported in [11] can be made smaller, chosen as the value reported in Algorithm 3, Line 2 and shown in Table 1. However, these conclusions are supported by Lemma 2.13, which depends on Theorem 2.6 and the incorrect bound on the probability. As a result, the reported precision of the algorithm is overestimated compared to the true precision.

5.2 Comparison to Precise Channel Capacity

We compare ApproxFlow with the SharpCDCL-based technique proposed by Klebanov et al. [21]. Both techniques follow the same steps: 1) generating a SAT formula from a program using CBMC [15], 2) specifying a projection scope, and 3) performing projected model counting (precise or approximate). We compare only the step where ApproxFlow differs from Klebanov et al.'s approach, namely projected model counting. For this comparison, we feed a SAT formula to both tools, along with a projection scope extracted from a C program. If SAT formulas are directly available from the existing benchmarks, we reuse those formulas, otherwise we generate them with CBMC.

To produce the SAT formulas we use a 32-bit CBMC version 5.6 with the arguments `--32 --dimacs --function function-name --unwind loop-bound`. The parameter `function-name` specifies the function containing the variables for which we want to measure channel capacity. The parameter `loop-bound` is the loop unrolling depth, set to 32. We insert projection scopes corresponding to the SAT variables in the formats required by sharpCDCL and ApproxMC2.

Finally, we run sharpCDCL and ApproxMC2 each with a timeout of 2 hours, unless otherwise specified. We measure the running time and the number of models reported by each tool. As explained earlier, the base-2 logarithm of the number of models gives us the channel capacity. If sharpCDCL times out, it reports a lower bound on the number of models it found, while ApproxMC2 does not currently have this feature implemented. Consequently, we report a lower bound only for sharpCDCL, when applicable. We run sharpCDCL with arguments `-countMode=2 -projection=projection-scope`, where `projection-scope` refers to a file containing the projection variables, and a `countMode` of 2 tells sharpCDCL to perform model counting, rather than just SAT-solving. We ran ApproxMC2 with no arguments, as the projection scope is specified as comments in the SAT formula file. The default tolerance ε for ApproxMC2 is 0.8 (~ 0.8 bits of error), and the default confidence is 80% ($\delta = 0.2$).

While we recognize that a large number of runs for each experiment would be ideal for statistical evidence with respect to running time, many of our experiments are long-running and doing this was not feasible. Therefore, our figures represent the results of a single invocation of each tool.

Table 1. Leakage reported by ApproxMC2 and sharpCDCL as a number of bits, relative error (as a percentage) in number of bits of channel capacity, running times for each tool, and speedup factor observed when running ApproxMC2 instead of sharpCDCL for several benchmarks. Negative entries represent slowdown factors. Speedup entries marked as — represent entries for which at least one of the tools completed too quickly for the precision of our timing tool (and at least one reported 0.00s). Entries marked with **error** represent values for which sharpCDCL terminated with an error, and could not produce a value for the model count. We note that in many cases, only ApproxMC2 was able to complete, with bolded entries representing cases in which both tools ran to completion. We note that ApproxMC2 never produced an error.

Benchmarks from [26, 5, 25, 21]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
e-purse	**5.00**	**5.00**	**0%**	**0.06**	**0.28**	**-4.67**
pw-checker	**1.00**	**1.00**	**0%**	**0.00**	**0.00**	—
sum-query	>22.49	32.00	*	t/o	0.87	*
10random	**3.32**	**3.32**	**0%**	**0.00**	**0.00**	—
bsearch16	**16.00**	**16.00**	**0%**	**3.40**	**0.49**	**6.90**
bsearch32	>22.87	32.00	*	t/o	2.13	*
mix-dupl	**16.00**	**16.00**	**0%**	**5.91**	**0.20**	**29.60**
sum32	>22.48	32.00	*	t/o	0.89	*
illustr.	**4.09**	**4.09**	**0%**	**0.00**	**0.01**	—
mask-cpy	**16.00**	**16.00**	**0%**	**6.02**	**0.20**	**30.1**
sanity-1	>22.82	31.04	*	t/o	0.94	*
sanity-2	>22.92	31.00	*	t/o	1.07	*
check-cpy	>22.51	32.00	*	t/o	0.88	*
copy	>22.49	32.00	*	t/o	0.84	*
div-by-2	>22.79	31.00	*	t/o	1.06	*
implicit	**>2.81**	**2.81**	**0%**	**0.00**	**0.01**	—
mul-by-2	>22.46	31.00	*	t/o	0.89	*
popcnt	**5.04**	**5.04**	**0%**	**0.00**	**0.01**	—
simp-mask	**8.00**	**8.00**	**0%**	**0.00**	**0.05**	—
switch	**4.25**	**4.25**	**0%**	**0.00**	**0.00**	—
tbl-lookup	>22.45	32.00	*	t/o	0.88	*

5.3 Benchmarks

Several benchmarks have become accepted in the QIF literature. Tables 1 to 3 show the relative error and speedup factor of running ApproxMC2 instead of sharpCDCL on these benchmarks. As no SAT formulas were openly available for many of these, we wrote C implementations from the descriptions of the specified benchmarks in their respective papers, and obtained SAT formulas using CBMC as previously described.

Table 1 ApproxMC2 and sharpCDCL on the benchmarks presented in [26, 5, 25, 21]. When both ApproxMC2 and sharpCDCL terminate before the time out, the reported model count is identical, therefore ApproxMC2 has relative error of zero. In the cases when sharpCDCL times out after two hours, the lower-bound channel capacity reported by sharpCDCL ranges from 22 to 23 bits, even when the actual result is larger. On these benchmarks, ApproxMC2 is not much slower than sharpCDCL, and always reports the exact result. The converse tells a different tale, with sharpCDCL often timing out after 2 hours, and providing only a coarse lower bound in these cases. It is also somewhat surprising that sharpCDCL times out on SAT formulas resulting from some simple programs, such as divide-by-2.

Table 2. Leakage reported by ApproxMC2 and sharpCDCL as a number of bits, relative error (as a percentage) in number of bits of channel capacity, running times for each tool, and speedup factor observed when running ApproxMC2 instead of sharpCDCL for several benchmarks. Negative entries represent slowdown factors. Speedup entries marked as — represent entries for which at least one of the tools completed too quickly for the precision of our timing tool (and at least one reported 0.00s). Entries marked with error represent values for which sharpCDCL terminated with an error, and could not produce a value for the model count. We note that ApproxMC2 never produced an error, and further note that in many cases, only ApproxMC2 was able to complete, with bolded entries representing cases in which both tools ran to completion. The entry fx was run with a higher timeout (8.5 hours) instead of the usual 2 hours.

Benchmarks from [26, 5, 25, 21]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
ddp	error	128.00	*	error	23.50	*
ddp.pp	error	128.00	*	error	19.55	*
popcount	**5.04**	**5.04**	**0%**	**0.00**	**0.01**	—
sanitize	**4.00**	**4.00**	**0%**	**0.00**	**0.00**	—
openssl.1	**8.00**	**8.00**	**0%**	**1.44**	**70.66**	**-49.10**
openssl.2	**16.00**	**16.00**	**0%**	**4.63**	**75.39**	**-16.30**
openssl.3	>22.24	24.00	*	t/o	92.47	*
openssl.4	>22.91	32.00	*	t/o	86.32	*
openssl.5	>23.10	40.00	*	t/o	87.74	*
openssl.6	error	48.00	*	error	89.60	*
openssl.7	error	56.00	*	error	91.98	*
openssl.8	error	64.00	*	error	98.04	*
openssl.9	error	72.00	*	error	97.41	*
openssl.10	error	80.00	*	error	112.71	*
openssl.15	error	t/o	*	error	t/o	*
openssl.20	error	160.00	*	error	142.48	*
swirl	>12.82	t/o	*	t/o	t/o	—
10random	**3.32**	**3.32**	**0%**	**0.00**	**0.01**	—
bsearch16	**16.00**	**16.00**	**0%**	**4.16**	**0.68**	**6.12**
bsearch16.pp	**16.00**	**16.00**	**0%**	**3.73**	**0.35**	**10.70**
bsearch32	>22.79	32.00	*	t/o	3.21	*
bsearch32.pp	>22.90	32.00	*	t/o	6.93	*
fx	**16.00**	**16.00**	**0%**	**5753.42**	**7307.61**	**-1.27**
mixdup	**16.00**	**16.00**	**0%**	**8.44**	**0.22**	**38.40**
sum.32	>22.78	32.00	*	t/o	0.98	*

In Table 2, we present results for a set of benchmarks described in [22, 21], for which the authors kindly provided us the SAT formulas directly. As in the previous set of experiments, when both ApproxMC2 and sharpCDCL report a number of models, the numbers are identical despite ApproxMC2's fairly relaxed theoretical tolerance and confidence. In these experiments, we found that sharpCDCL sometimes incorrectly terminates before its timeout because of two kinds of error: a segmentation fault, or reporting the formula to be unsatisfiable (despite normally giving a lower bound on the number of solutions if interrupted). In addition, we witness cases in which ApproxMC2 timed out. We observe that ApproxMC2 is slower than sharpCDCL on short-running experiments (openssl.1 and openssl.2), but significantly faster on the more difficult, longer-running experiments (where sharpCDCL often times out), with the exception of fx.

In Table 3, we present results for a set of scalable case studies given in [9]. These case studies consist of two models of a Voting protocol (one based on each voter voting for a single-candidate, and one based on each voter having a

Table 3. Leakage reported by ApproxMC2 and sharpCDCL as a number of bits, relative error (as a percentage) in number of bits of channel capacity, running times for each tool, and speedup factor observed when running ApproxMC2 instead of sharpCDCL for several benchmarks. Negative entries represent slowdown factors. Entries marked with error represent values for which sharpCDCL terminated with an error, and could not produce a value for the model count (even in cases it completed within the timeout, it did not report a number of solutions). We note that ApproxMC2 never produced an error, with bolded entries representing cases in which both tools ran to completion. The entries with 0% error had a reported solution count of 0 by both tools, so we abuse notation and consider this a 0%, rather than undefined, error.

Benchmarks from [26, 5, 25, 21]

Experiment name	sharpCDCL leakage	ApproxMC2 leakage	Relative error	sharpCDCL time	ApproxMC2 time	Speedup factor
Sing.3	error	5.81	*	error	1.46	*
Sing.5	**7.62**	**7.86**	**3.15%**	**0.06**	**3.02**	**-50.30**
Sing.7	**9.63**	**9.70**	**0.73%**	**0.38**	**3.98**	**-10.50**
Sing.9	**10.97**	**11.00**	**0.27%**	**0.83**	**5.82**	**-7.01**
Rank.3	>21.00	67.17	0%	t/o	55.34	*
Rank.5	**0.00**	**0.00**	**0%**	**0.40**	**0.52**	**-1.30**
Rank.7	**0.00**	**0.00**	**0%**	**0.75**	**0.96**	**-1.28**
Rank.9	**0.00**	**0.00**	**0%**	**1.26**	**1.58**	**-1.25**

preference ranking of the candidates). These experiments have parameters that control the size of the program, and therefore of the generated SAT formula. We refer the reader to [9] for a description of the case studies and their parameters. We translated the Java code provided on the companion website of the paper into C, and generated SAT formulas with CBMC, with 16 as the bound for loop unwinding. The experiment names beginning with "Sing" represent the single candidate case from the case studies, while the experiment names beginning with "Rank" represent the preference ranking case. In both cases, we correspond cases in which ApproxMC2 is not clearly better than sharpCDCL. Although sharpCDCL produces an error or times out in two of these cases, when sharpCDCL terminates, it is between 7 and 50 times faster than ApproxMC2. We believe this is because the resulting SAT formulas are dense in the number of solutions, which is a weakness of ApproxMC2 (as we stated in Section 4.2). Nonetheless, ApproxMC2 is very precise, exhibiting relative errors ranging from 0.30% to 3.10%. The Rank entries with the number of candidates ranging from 5 to 9 represent unsatisfiable formulas, and thus have 0 solutions.

As a consequence of the errors returned by sharpCDCL and the number of benchmarks for which ApproxMC2 reported the exact count, we lack an in-depth empirical evaluation of ApproxMC2's precision. To this end, we present further relative error measurements on the SmartGrid benchmark from [9]. These benchmarks compute the leakage of private information obtained by observing the global energy consumption in a smart grid. One model computes the information about a single house, and the other computes the information about the consumption of every house. As in the Voting protocol, we can scale the benchmark by changing the values of the protocol's parameters (Case A or B), the number of houses, and (for the single-house case), the size of the house – small (S), medium (M), or large(L).

We refer the reader to [9] for the full details of these models and their parameters. We present in Table 4 the relative error percentage of ApproxMC2 with respect to sharpCDCL, on the number of bits of leakage reported. As we can see, the channel capacity reported by the tools was very close (and in many cases exactly equal) in all cases when both tools ran to completion and reported a figure, except for case A, N=36 of the global leakage experiment, where we see an "error" of 45.25% when compared to sharpCDCL. This large error results from an incorrect channel capacity measurement reported by sharpCDCL. We verified this using the exact projected model counter SharpSubSAT from [31], observing a relative error of 2.86% in the channel capacity when compared to this counter.

Table 4. Relative error (as a percentage) in the channel capacity estimation by using ApproxMC2 instead of the precise counter sharpCDCL for the SmartGrid case study from [9]. The entry marked as — represents a case in which sharpCDCL returned an error and could not report a result. The entry marked with a * represents a case in which sharpCDCL returned an incorrect channel capacity (resulting in an observed 45.25% relative error). We compared ApproxMC2 to another exact counter (SharpSubSAT [31]) which reported the correct precise value, to obtain the 2.86% figure.

		Relative error			
	Num	**Single house**			**Global**
Case	**houses**	**S**	**M**	**L**	
A	**36**	0.32%	0.32%	0.32%	2.86%*
A	**49**	0.00%	0.00%	0.00%	0.00%
A	**64**	0.31%	0.31%	0.31%	0.32%
B	**36**	0.20%	0.58%	0.20%	—
B	**49**	0.26%	0.26%	0.26%	0.26%
B	**64**	0.10%	0.10%	0.29%	0.10%

Finally, we remark that, in addition to being much faster in most cases while maintaining very high precision, ApproxMC2 is able to report an approximate model count in all our experiments, in contrast to the significant number of error cases reported by sharpCDCL.

Comparison to ApproxMC-P

Although the work presented in [22] suffers from the theoretical errors that we described in Section 1, we compared against the implementation of their algorithm, called ApproxMC-P. We repeated the experiments from Section 5.3 for the Voting and SmartGrid case studies, using ApproxMC-P with the cryptominisat4 [30] backend, instead of ApproxMC2. We used the same values of ϵ and δ as in Section 1, but ran with a timeout of only 5 minutes instead of 2 hours. We found that in all but 2 cases, the tool reported a spurious model count of 0 (in those two non-zero cases, ApproxMC-P reported the exact count). We also tried using the sharpCDCL backend instead of the cryptominisat4 backend, and results were similar, with most cases resulting in an error. Additionally, we also ran on other experiments described in Section 5, observing a high occurrence of 0 reported as the model count. We similarly omit these due to space constraints.

```
1   int dtls1_process_heartbeat(SSL *s) {
2
3     unsigned char *p = &s->s3->rrec.data[0], *pl;
4     unsigned short hbtype;
5     unsigned int payload;
6     unsigned int padding = 16;
7     //...
8     hbtype = *p++;
9     n2s(p, payload);
10
11    if (1+2 + payload+16 > s->s3->rrec.length)
12      return 0; /* missing in bugged version */
13
14    if (hbtype == TLS1_HB_REQUEST) {
15      unsigned char *buffer, *bp;
16      unsigned int write_length =
17         1 + 2 + payload + padding;
18      //..
19      buffer = OPENSSL_malloc(write_length);
20      bp = buffer;
21      *bp++ = TLS1_HB_RESPONSE;
22      s2n(payload, bp);
23      memcpy(bp, pl, payload);
24      //send buffer ...
25    }
26  }
```

(a)

```
1   int dtls1_process_heartbeat(char* input_msg,
2                               int   msg_len){
3     char *p = input_msg;
4     unsigned short hbtype;
5     unsigned int payload ;
6     unsigned int padding = 0; // ignore padding
7     hbtype = *p;
8     p++;
9     n2s(p,payload);
10
11    // only present in model for correct version
12    __CPROVER_assume(1 + 2 + payload <= msg_len);
13
14    // we model only the if true branch
15    unsigned char buffer[3 + MAX_PAYLOAD_SIZE];
16    unsigned char *bp;
17
18    set_to_zero(buffer, 3 + MAX_PAYLOAD_SIZE);
19    bp = buffer;
20    *bp = TLS1_HB_RESPONSE;
21    bp++;
22    s2n(payload,bp);
23    memcpy_emul(bp,p,payload);
24
25    return 0;
26  }
```

(b)

Fig. 3. Code model for the Heartbleed bug. a) Simplified fragment of code from `ssl/d1_both.c` in OpenSSL 1.0.1f. b) Model for analysis.

6 Case Study: Heartbleed Bug

We present a case study for our technique based on the Hearbleed OpenSSL bug [1]. We show that ApproxFlow can handle the complexity required to detect the bug, in contrast to the state of the art of precise QIF.

The Heartbleed bug. The Heartbleed bug [1] is a vulnerability in the OpenSSL implementation of the Heartbeat extension of TLS and DTLS [2]. It was introduced in the OpenSSL code in 2012, and discovered and patched between March and April 2014. It has been estimated that at discovery time between 24% and 55% of the HTTPS servers in the Alexa Top 1 Million list were vulnerable to it [18]. The fact that Heartbleed went unnoticed for 2 years led the security development community to ask why the automated techniques used to scan the OpenSSL code for vulnerabilities did not detect it earlier, and which static and dynamic techniques could be expected to find bugs similar to Heartbleed [28, 33, 35]. We show how QIF can be used to model and detect the Heartbleed bug.

Fundamentally, the bug consists of a buffer over-read on a `memcpy()` function call in the Heartbeat implementation in OpenSSL, specifically in function `dtls1_process_heartbeat()` of file `d1_both.c`. Figure 3 (a) presents a fragment of the function. To verify that a server is still functional, the Heartbeat protocol has the client ask the server to reply with a specific word. In the OpenSSL implementation, the chosen word and its length are under the control of the client. The length of the word is encoded in the first bytes of the message. The pointer `p` is set at the start of the message from the client passed to the function via the `SSL` structure in argument (line 3). First, the function decodes the type and length of the message and stores it in the `payload` variable (line 8-9). In the

vulnerable version, the checking of `payload` (line 11) is absent, which allows the client to specify a word length greater than the actual length of the sent word. Next, the function allocates a buffer large enough to store the answer message for the client (line 19). A call to `memcpy()` (line 23) fills that buffer with the input word and, if the value of `payload` is greater than the length of the input word, the content of the memory after the input word. Finally, `buffer` is sent back to the client. With the bugged version, the client can obtain restricted kernel memory, which they can use to infer privileged information about the server, e.g. the server's private key.

Modeling the bug. We had to rewrite OpenSSL C code in a different form due to limitations in the currently-available tools that can produce SAT formulas from C code. This modelling step is not inherent in our approach, and may largely disappear in the long term as SAT-formula-generation tools mature. Generating the formulas from C code is out of the scope of this paper, and we rely on CBMC to perform this transformation. Therefore, the code that we actually analyzed is a model of the real code – one that CBMC can handle.

Our model is presented in Fig. 3 (b). Our goal is to compare the channel capacity of the `input_msg` and `buffer` arrays. Since calls to `malloc` are not well-supported by CBMC, we statically allocate the array (line 15). By default, CBMC considers that unassigned values are unconstrained, therefore we set each cell of `buffer` to zero with the `set_to_zero` macro on line 16. We then fill the `buffer` as in the original function, but instead of calling `memcpy()`, we invoke on line 21 a macro `memcpy_emul` that uses a loop to copy the values.

In order to statically set the size of `buffer`, we need to know the maximum value taken by the variable `payload`. This variable is encoded by 16 bits. However, we restrict it by adding CBMC constraints on `input_msg` so that we choose the number of bits. The constant `MAX_PAYLOAD_SIZE` is set accordingly. For the experiments, we restrict the value of `payload` to be encoded by 4 bits, which corresponds to a message of at most 15 bytes. We set the message length to 1 byte, as an attacker would do to maximize the amount of information obtained from the memory.

Due to another limitation in CBMC, we were not able to analyze the if-error-then-return idiom, replacing it with a CBMC assumption negating the condition of the if statement. Similarly, we only modeled the true branch of the second conditional.

Results. We first analyze the model of the vulnerable version, that is without the CBMC assumption about `payload` on line 12. Executing CBMC on the model in Fig. 3 (b) produces a SAT formula with 39272 clauses in less than 1 second. Since our `memcpy` is implemented by a loop on `payload`, we set the bounds on the loop to 260 (instead of 32 as in the benchmarks from Section 5), a figure chosen due to CBMC limitations.

First, we measure the channel capacity to `input_msg`. Both sharpCDCL and ApproxMC2 terminate in less than a second and return 12 bits, which correspond to 4 bits to encode the size of the message and 1 byte for the message itself.

We then measure the channel capacity to `buffer`. The sharpCDCL tool times out after 2 hours of trying to count the models in the formula. On the other hand, the ApproxMC2 tool provides an approximate channel capacity of 15 bytes in 25 seconds. Since the channel capacity to `buffer` is much more than the one to `input_msg`, there is a suspicious leak of approximately 14 bytes of information. By reducing the confidence to 50% ($\delta = 0.5$), ApproxMC2 returns 15.1 bytes in 2 seconds. After such analysis, a programmer could investigate why this leakage is so high and possibly discover the Heartbleed bug.

When adding the CBMC assumption representing the patch to fix the bug on line 12, the leakage of `buffer` is down to about 1 byte (257 models) and both sharpCDCL and ApproxMC2 terminate in less than a second. ApproxMC2 reports 264 models. This leakage value indicates that, as expected, the buffer actually transmits one byte of information and that the patch successfully removed the suspicious leak.

7 Discussion and Future Work

In this section, we discuss the broader meaning of our approach, its limitations, and provide discourse on the results of the evaluation in Section 5, as well as discussion on future directions.

We showed in Section 5 that an approximate approach can provide a large increase in performance at the cost of a small amount of precision, especially as problem sizes increase. A major strength of ApproxFlow is its ability to trade efficiency for precision simply by varying the tolerance parameter ε. The ability to relax the precision to a desired level can yield practical results in many cases. Consider a program meant to return a value from a small set of return codes. The corresponding leakage might be only 1 or 2 bits. In this case, a coarse approximation would be sufficient; an observation of *approximately* 10 bits is just as practically significant as an observation of *precisely* 10 bits – both would mark the program as suspicious, prompting further analysis.

In Section 6, we showed that with ApproxFlow, we can perform a largely automated analysis which is potentially useful in discovering, or confirming, bugs in real software. Nonetheless, we recognize the need for improvements to the technique before we can realize a fully-automated and practically useful bug-finding tool. As explained in Section 6, limitations in CBMC force us to analyze manually-simplified versions of some programs.

A possible improvement to formula generation would be to pursue source code in a language easier to analyze than C. Higher-level languages such as Java or C# present easier analysis for model checkers and symbolic execution engines, because of features such as stronger type-checking. While C is arguably still the most relevant language for targeting security bugs, it is perhaps too ambitious a target for current formula generation techniques. Since the formula generation is decoupled from the model counting, it would be interesting to study the effectiveness of our overall approach for Java or C# source code, using a tool such as Java PathFinder as its formula generation engine.

In [26], Newsome et al. present the use of channel capacity measurement as a way to enforce k-bit policies, which are policies of the form *"the program leaks no more than k bits of information from its inputs to its outputs."* Such policies may be used as an aid to a developer looking for security issues in source code – as soon as "too many" bits are found, the offending part of the program can be flagged as suspicious. This is a natural use case for approximation, as the lower bound is often already a fuzzy quantity, and a choice for the value of k may be somewhat arbitrary. As a future direction, it would be interesting to use an approximate *lower bounding* projected model counter, and observe its efficacy compared to ApproxMC2 for enforcing k-bit policies. This use case, for example, gives *quantitative* information flow techniques (such as our method) a distinct advantage over qualitative ones, which do not reason about the *size* of the flow.

Finally, it would be illuminating to compare our technique to the MaxCount tool presented by Fremont et al. [19]. Using the underlying approximate counting algorithm they present in the place of ApproxMC2, we could study how sensitive ApproxFlow is to the choice of counting algorithm. We expect that an approach based on MaxCount might be more effective than our own for large leaks (relative to the formula size), but not for small leaks. Perhaps a combination of the two counting algorithms would be the most effective in practice.

8 Conclusions

We have presented ApproxFlow, a technique leveraging approximate model counting to measure the approximate channel capacity of deterministic C programs, showing it to be among the most efficient currently-available techniques for QIF computation. The necessity of such a technique arises from both theoretical errors and practical limitations in some of the prior work that applied approximate model counting to channel capacity measurement.

ApproxFlow takes a program, performs model checking to produce a formula which represents the program, and leverages approximate projected model counting in order to obtain an approximation of the program's channel capacity. We show how ApproxFlow is more efficient than state-of-the-art techniques on a number of benchmarks, with graceful degradation in the relatively few cases when it's less efficient. In particular, on many benchmarks, we show that ApproxFlow can estimate the information flow while precise tools cannot, or otherwise obtain significant speedups while maintaining high empirical precision, and exhibiting much smaller slowdown factors when ApproxFlow is slower.

In addition, we present a new case study based on the famous OpenSSL Heartbleed bug that showcases the power of our technique. While analysis with state-of-the-art precise tools times out after 2 hours, ApproxFlow obtains the channel capacity in only 25 seconds.

Our technique opens up the possibility of automatically detecting channel capacity for larger programs than previously possible, representing a step towards automatic vulnerability detection using QIF.

References

1. CVE-2014-0160 "Heartbleed". `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`. Accessed: 2017-04-03.
2. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. `https://tools.ietf.org/html/rfc6520`. Accessed: 2017-04-03.
3. R. A. Aziz, G. Chu, C. J. Muise, and P. J. Stuckey. #∃SAT: Projected model counting. In *Theory and Applications of Satisfiability Testing, SAT 2015, 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, pages 121–137, 2015.
4. D. Babic. Satisfiability Suggested Format. Technical report, 015 1993.
5. M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 141–153, 2009.
6. M. Bellare, O. Goldreich, and E. Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.
7. F. Biondi, Y. Kawamoto, A. Legay, and L. Traonouez. Hyleak: Hybrid analysis tool for information leakage. In D. D'Souza and K. N. Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 156–163. Springer, 2017.
8. F. Biondi, A. Legay, P. Malacaria, and A. Wasowski. Quantifying information leakage of randomized protocols. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 68–87, 2013.
9. F. Biondi, A. Legay, and J. Quilbeuf. Comparative analysis of leakage tools on scalable case studies. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, pages 263–281, 2015.
10. F. Biondi, A. Legay, L. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 702–707, 2013.
11. S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pages 200–216, 2013.
12. S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 3569–3576, 2016.
13. D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. *Electr. Notes Theor. Comput. Sci.*, 59(3):238–251, 2001.
14. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking.* MIT Press, 2001.
15. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004, Barcelona, Spain, March 29 - April 2, 2004*, pages 168–176, 2004.

16. T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., 1991.
17. D. E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.
18. Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.
19. D. J. Fremont, M. N. Rabe, and S. A. Seshia. Maximum model counting. In S. P. Singh and S. Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 3885–3892. AAAI Press, 2017.
20. R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, 1989.
21. V. Klebanov, N. Manthey, and C. J. Muise. SAT-based analysis and quantification of information flow in programs. In *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, pages 177–192, 2013.
22. V. Klebanov, A. Weigl, and J. Weisbarth. Sound probabilistic #SAT with projection. In *Proceedings 14th International Workshop Quantitative Aspects of Programming Languages and Systems, QAPL 2016, Eindhoven, The Netherlands, April 2-3, 2016.*, pages 15–29, 2016.
23. P. Malacaria, M. Tautchning, and D. Distefano. Information leakage analysis of complex C code and its application to openssl. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pages 909–925, 2016.
24. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI 2008, Tucson, AZ, USA, June 7-13, 2008*, pages 193–205, 2008.
25. Z. Meng and G. Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011, San Jose, CA, USA, 5 June, 2011*, page 1, 2011.
26. J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 73–85, 2009.
27. Q. Phan and P. Malacaria. Abstract model counting: a novel approach for quantification of information leaks. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 283–292, 2014.
28. J. Sass. The role of static analysis in Heartbleed. `https://www.sans.org/reading-room/whitepapers/threats/role-static-analysis-heartbleed-35752`. Accessed: 2017-04-03.
29. J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.*, pages 331–340, 1993.
30. M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th*

International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings, pages 244–257, 2009.
31. C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu. Precisely measuring quantitative information flow: 10K lines of code and beyond. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 31–46, 2016.
32. L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 8(3):410–421, 1979.
33. J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu. Risk assessment of buffer "Heartbleed" over-read vulnerabilities. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pages 555–562, 2015.
34. A. Weigl. Efficient SAT-based pre-image enumeration for quantitative information flow in programs. In *Data Privacy Management and Security Assurance - 11th International Workshop, DPM 2016 and 5th International Workshop, QASA 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings*, pages 51–58, 2016.
35. D. A. Wheeler. How to prevent the next Heartbleed. `https://www.dwheeler.com/essays/heartbleed.html`. Accessed: 2017-04-03.

Code Obfuscation Against Abstract Model Checking Attacks

Roberto Bruni[1], Roberto Giacobazzi[2,3], and Roberta Gori[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Dipartimento di Informatica, Università di Verona, Italy
[3] IMDEA SW Institute, Spain

Abstract. Code protection technologies require anti reverse engineering transformations to obfuscate programs in such a way that tools and methods for program analysis become ineffective. We introduce the concept of model deformation inducing an effective code obfuscation against attacks performed by abstract model checking. This means complicating the model in such a way a high number of spurious traces are generated in any formal verification of the property to disclose about the system under attack. We transform the program model in order to make the removal of spurious counterexamples by abstraction refinement maximally inefficient. A measure of the quality of the obfuscation obtained by model deformation is given together with a corresponding best obfuscation strategy for abstract model checking based on partition refinement.

1 Introduction

Software systems are a strategic asset, which in addition to correctness deserves security and protection. This is particularly critical with the growth of mobile computing, where the traditional black-box security model, with the attacker not able to see into the implementation system, is not anymore adequate. Code protection technologies are increasing their relevance due to the ubiquitous nature of modern untrusted environments where code runs. From home networks to consumer devices (e.g., mobile devices, cloud, and IoT devices), the running environment cannot guarantee integrity and privacy. Existing techniques for software protection originated with the need to protect license-checking code, particularly in games or in IP protection. Sophisticated techniques, such as white-box (WB) cryptography and software watermarking, were developed to prevent adversaries from circumventing anti-piracy protection in digital rights management systems.

A WB attack model to a software system $\mathcal{S}$ assumes that the attacker has full access to all of the components of $\mathcal{S}$, i.e., $\mathcal{S}$ can be inspected, analysed, verified, reverse-engineered, or modified. The goal of the attack is to disclose properties of the run-time behaviour of $\mathcal{S}$. These can be a hidden watermark [22,18,9], a cryptographic key or an invariance property for disclosing program semantics and make correct reverse engineering [7]. Note that standard encryption is only partially applicable for protecting $\mathcal{S}$ in this scenario: The transformed code has to be executable while being protected. Protection is therefore implemented

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 94–115, 2018.
https://doi.org/10.1007/978-3-319-73721-8_5

as *obfuscation* [6]. Essentially, an obfuscator is a compiler that transforms a program p into a semantically equivalent one $\mathcal{O}(p)$ but harder to analyse and reverse engineer. In many cases it is enough to guarantee that the attacker cannot disclose the information within a bounded amount of time and with limited resources available. This is the case if new releases of the program are issued frequently or if the information to be disclosed is some secret key whose validity is limited in time, e.g., when used in pay-per-view mobile entertainment and in streaming of live events. Here the goal of the code obfuscation is to prevent the attacker from disclosing some keys before the end of the event.

The current state of the art in code protection by obfuscation is characterised by a scattered set of methods and commercial/open-source techniques employing often ad hoc transformations; see [7] for a comprehensive survey. Examples of obfuscating transformations include code flattening to remove control-flow graph structures, randomised block execution to inhibit control-flow graph reconstruction by dynamic analysis, and data splitting to obfuscate data structures. While all these techniques can be combined together to protect the code from several models of attack, it is worth noting that each obfuscation strategy is designed to protect the code from one particular kind of attack. However, as most of these techniques are empirical, the major challenges in code protecting transformations are: (1) the design of provably correct transformations that do not inject flaws when protecting code, and (2) the ability to prove that a certain obfuscation strategy is more effective than another w.r.t. some given attack model.

In this paper we consider a quite general model of attack, propose a measure to compare different obfuscations and define a best obfuscation strategy.

The aim of any attack is to disclose some program property. It is known that many data-flow analyses can be cast to model checking of safety formulas. For example, computing the results of a set of data-flow equations is equivalent to computing a set of states that satisfies a given modal/temporal logic specification [20,21]. Even if several interesting properties are not directly expressed as safety properties, because they are existentially quantified over paths, their complements are relevant as well and are indeed safety properties, i.e. they are requested to hold for all reachable states. For these reasons $\forall$CTL* is a suitable formal logic to express those program properties the attacker wants to disclose.

In this context, program analysis is therefore the model checking of a $\forall$CTL* formula on a(n approximate) model associated with the program. The complexity of software analysis requires automated methods and tools for WB attack to code. Since the attacker aims to disclose the property within a bounded time and using bounded resources, approximate methods such as abstract interpretation [8] or abstract model checking [3] are useful to cope with the complexity of the problem. The abstraction here is helpful to reduce the size of the model keeping only the relevant information which is necessary for the analysis. Safety properties expressed in $\forall$CTL* can be model-checked using abstraction refinement techniques (CEGAR [2]) as in Fig. 1. An initial (overapproximated) abstraction of the program is model-checked against the property ϕ. If the verification proves that ϕ holds true, then it is disclosed. Similarly, if an abstract

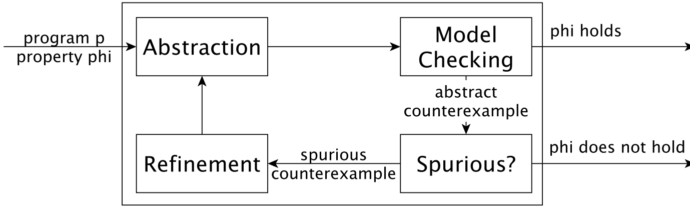


Fig. 1. Counterexample guided abstraction refinement (CEGAR) framework

counterexample is found that corresponds to a concrete counterexample, it is disclosed that ϕ is not valid. An abstract counterexample that is present in the existential overapproximation but not in the concrete model is called *spurious*. If a spurious counterexample is found the abstraction is refined to eliminate it and the verification is repeated on the refined abstraction. Of course, the coarser the abstraction that can be used to verify the property the more effective the attack is. Indeed, the worst case for the attacker is when the verification cycle must be repeated until the refined abstraction coincides with the concrete model.

Given a program p and a $\forall$CTL* property ϕ to be obfuscated we aim to:

1. define a measure to compare the effectiveness of different obfuscations of p;
2. derive an optimal (w.r.t. the above measure) obfuscation of p.

The measure of obfuscation that we propose is based on the size of the abstract model that allows the attacker to disclose the validity of ϕ. Intuitively, the larger the size of the model, the more resources and computation power the attacker needs to spend to reach the goal.

We propose a systematic model deformation that induces a systematic transformation on the program (obfuscation). The idea is to transform the source program in such a way that:

1. its semantics is preserved: the model of the original program is isomorphic to the (reachable) part of the model of the obfuscated program (Theorem 1);
2. the performance is preserved;
3. the property ϕ is preserved by the transformation (Theorem 2);
4. such transformation forces the CEGAR framework to ultimately refine the abstract model of the transformed program into the concrete model of the original program (Theorem 3). Therefore any abstraction-based model checking becomes totally ineffective on the obfuscated program.

CEGAR can be viewed as a learning procedure, devoted to learn the partition (abstraction) which provides a (bisimulation) state equivalence. Our transformation makes this procedure extremely inefficient. Note that several instances of the CEGAR framework are possible depending on the chosen abstraction and refinement techniques (e.g. predicate refinement, partition refinement) and that CEGAR can be used in synergy with other techniques for compact representation of the state space (e.g., BDD) and for approximating the best refined

abstraction (e.g., SAT solvers). Notably, CEGAR is employed in state-of-the-art tools as Microsoft's SLAM engine for the SDV framework [17]. Here we focus on the original formulation of CEGAR based on partition refinement, but we believe that our technique can be extended to all the other settings.

As many obfuscating transformations, our method relies on the concept of *opaque expressions* and *opaque predicate* that are expressions whose value is difficult for the attacker to figure out. Opaque predicates and expressions are formulas whose value is uniquely determined (i.e. a constant), independently from the parameters they receive, but this is not immediately evident from the way in which the formula is written. For example it can be proved that the formula $x^2 - 34y^2 \neq 1$ is always true for any integer values of x and y. Analogously, the formula $(x^2 + x)$ **mod** $2 \neq 0$ is always false. These expression/predicate are, in general, constructed using number properties that are hard for an adversary to evaluate. Of course such predicates can be parameterised so that each instance will look slightly different. Opaque predicates/expressions are often used to inject dead code in the obfuscated program in such a way that program analysis cannot just discard it. For example, if the guard $x^2 - 34y^2 \neq 1$ is used in a conditional statement, then the program analysis should consider both the "then" and the "else" branches, while only the first is actually executable. In this paper: i) opaque expressions will be used to hide from the attacker the initial values of the new variables introduced by our obfuscation procedure; and ii) opaque predicates will be used to add some form of nondeterminism originated from model deformations. The effects of the opaque expressions and predicates will be similar: since the attacker will not be able to figure out their actual values, all the possible values have to be taken into account.

Plan of the paper. In Section 2 we recall CEGAR and fix the notation. In Section 3 we introduce the concept of model deformation and define the measure of obfuscation. In Section 4 we define a best obfuscation strategy. Our main results are in Section 5. Some concluding remarks are in Section 6

Related works. With respect to previous approaches to code obfuscation, all aimed to defeat attacks based on specific abstractions, we define the first transformation that defeats the refinement strategy, making our approach independent on the specific attack carried out by abstract model checking.

Most existing works dealing with practical code obfuscation are motivated by either empirical evaluation or by showing how specific models of attack are defeated, e.g., decompilation, program analysis, tracing, debugging (see [7]). Along these lines, [23] firstly considered the problem of defeating specific and well identified attacks, in this case control-flow structures. More recently [1] shows how suitable transformations may defeat symbolic execution attacks. We follow a similar approach in defeating abstract model-checking attacks by making abstraction refinements maximally inefficient. The advantage in our case is in the fact that we consider abstraction refinements as targets of our code protecting transformations. This allows us both to extract suitable metrics and to apply our transformations to *all* model checking-based attacks.

A first attempt to formalise in a unique framework a variety of models of attack has been done in [13,10,14] in terms of abstract interpretation. The idea is that, given an attack implemented as an abstract interpreter, a transformation is derived that makes the corresponding abstract interpreter incomplete, namely returning the highest possible number of false alarms. The use of abstract interpretation has the advantage of making it possible to include in the attack model the whole variety of program analysis tools. While this approach provides methods for understanding and comparing qualitatively existing obfuscations with respect to specific attacks defined as abstract interpreters, none of these approaches considers transformations that defeat the abstraction refinement, namely the procedure that allows to improve the attack towards a full disclosure of the obfuscated program properties.

Even if the relation between false alarms and spurious counterexamples is known [15] to the best of our knowledge, no obfuscation methods have been developed in the context of formal verification by abstract model checking, or more in general by exploiting structural properties of computational models and their logic.

2 Setting The Context

2.1 Abstract Model Checking

Given a set Prop of propositions, we consider the fragment $\forall$CTL* of the temporal logic CTL* over Prop [4,12]. Models are *Kripke structures* $\mathcal{K} = \langle \Sigma, R, I, \| \cdot \| \rangle$ with a *transition system* $\langle \Sigma, R \rangle$ having *states* in Σ and *transitions* $R \subseteq \Sigma \times \Sigma$, *initial states* $I \subseteq \Sigma$, and an *interpretation function* $\| \cdot \|$: $\mathsf{Prop} \longrightarrow \wp(\Sigma)$ that maps each proposition p to the set $\| \mathsf{p} \| \subseteq \Sigma$ of all and only states where p holds. For $\forall$CTL* the notion of *satisfaction* of a formula φ in $\mathcal{K}$ is as usual [11], written $\mathcal{K} \models \varphi$ A *path* in $\langle \Sigma, R, I, \| \cdot \| \rangle$ is an infinite sequence $\pi = \{s_i\}_{i \in \mathbb{N}}$ of states such that $s_0 \in I$ and for every $i \in \mathbb{N}$, $R(s_i, s_{i+1})$. Terminating executions are paths where the final state repeats forever. We will sometimes use π to denote also a finite path prefix $\{s_i\}_{i \in [0,n]}$ for some $n \in \mathbb{N}$. Given a path $\pi = \{s_i\}_{i \in \mathbb{N}}$ and a state $x \in \Sigma$, we write $x \in \pi$ if $\exists i \in \mathbb{N}$ such that $x = s_i$.

Any state partition $P \subseteq \wp(\Sigma)$ defines an abstraction merging states into abstract states, i.e., an *abstract state* is a set of concrete states and the abstraction function $\alpha_P : \Sigma \rightarrow \wp(\Sigma)$ maps each state s into the partition class $\alpha_P(s) \in P$ that contains s. The abstraction function can be lifted to a pair of adjoint functions $\alpha_P : \wp(\Sigma) \rightarrow \wp(\Sigma)$ and $\gamma_P : \wp(\Sigma) \rightarrow \wp(\Sigma)$, such that for any $X \in \wp(\Sigma)$, $\alpha_P(X) = \bigcup_{x \in X} P(x)$ [19]. When the partition P is clear from the context we omit the subscript. A partition P with abstraction function α induces an *abstract Kripke structure* $\mathcal{K}_P = \langle \Sigma^\sharp, R^\sharp, I^\sharp, \| \cdot \|^\sharp \rangle$ that has abstract states in $\Sigma^\sharp = P$, ranged by $s^\sharp$, and is defined as the existential abstraction induced by P:

- $R^\sharp(s_1^\sharp, s_2^\sharp)$ iff $\exists s, t \in \Sigma.\ R(s,t) \wedge \alpha(s) = s_1^\sharp \wedge \alpha(t) = s_2^\sharp$,
- $s^\sharp \in I^\sharp$ iff $\exists t \in I.\ \alpha(t) = s^\sharp$,
- $\| \mathsf{p} \|^\sharp \stackrel{\text{def}}{=} \{\ s^\sharp \in \Sigma^\sharp \mid s^\sharp \subseteq \| \mathsf{p} \|\ \}$,

An abstract path in the abstract Kripke structure $\mathcal{K}_P$ is denoted by $\pi^\sharp = \{s_i^\sharp\}_{i\in\mathbb{N}}$. The abstract path associated with the concrete path $\pi = \{s_i\}_{i\in\mathbb{N}}$ is the sequence $\alpha(\pi) = \{\alpha(s_i)\}_{i\in\mathbb{N}}$. Vice versa, we denote by $\gamma(\pi^\sharp)$ the set of concrete paths whose abstract path is $\pi^\sharp$, i.e., $\gamma(\pi^\sharp) = \{\ \pi \mid \alpha(\pi) = \pi^\sharp\ \}$.

A counterexample for φ is either a finite abstract path or a finite abstract path followed by a loop. Abstract model checking is sound by construction: *If there is a concrete counterexample for φ then there is also an abstract counterexample for it*. Spurious counterexamples may happen: *If there is an abstract counterexample for φ then there may or may not be a concrete counterexample for φ.*

2.2 Counter-Example Guided Abstraction Refinement

With an abstract Kripke structure $\mathcal{K}^\sharp$ and a formula φ, the CEGAR algorithm works as follows [2]. $\mathcal{K}^\sharp$ is model checked against the formula. If no counterexample to $\mathcal{K}^\sharp \models \varphi$ is found, the formula φ is satisfied and we conclude. If a counterexample $\pi^\sharp$ is found which is not spurious, i.e., $\gamma(\pi^\sharp) \neq \emptyset$, then we have an underlying concrete counterexample and we conclude that φ is not satisfied. If the counterexample is spurious, i.e., $\gamma(\pi^\sharp) = \emptyset$, then $\mathcal{K}^\sharp$ is further refined and the procedure is repeated. The procedure illustrated in Fig. 1 induces an abstract model-checker attacker that can be specified as follows in pseudocode.

```
Input: program p, property φ
P = init(p, φ);
K = kripke(P, p);
c = check(K, φ);
while (c != null && spurious(p, c)) {
    P = refine(K, p, c);
    K = kripke(P, p);
    c = check(K, φ); }
return ((c == null), P);
```

Here we denote by *init* a function that takes a program p and the property φ and returns an initial abstraction P (a partition of variable domains); a function *kripke* that generates the abstract Kripke structure associated with a program p and the partition P; a function *check* that takes an abstract Kripke structure K and a property φ and returns either **null**, if $K \models \varphi$, or a (deterministically chosen) counterexample c; a predicate *spurious* that takes the program p and an abstract counterexample c and returns true if c is a spurious counterexample and false otherwise; and a function $\mathit{refine}(K, p, c)$ that returns a partition refinement so to eliminate the spurious counterexample c. As the model is finite, the number of partitions that refine the initial partition is also finite and the algorithm terminates by returning a pair: a boolean that states the validity of the formula, and the final partition that allows to prove it.

If several spurious counterexamples exist, then the selection of one instead of another may influence the refinements that are performed. For example, the same refinement that eliminates a spurious counterexample may cause the disappearance of several other ones. However, all the spurious counterexamples must

be eliminated. When we assume that *check* is deterministic, we just fix a total order on the way counterexamples are found. For example, we may assume that a total order on states is given (e.g., lexicographic) and extend it to paths.

Central in CEGAR is partition refinement. The algorithm identifies the shortest prefix $\{s_i^\#\}_{i\in[0,k+1]}$ of the abstract counterexample $\pi^\sharp$ that does not correspond to a concrete path in the model. The second to last abstract state $s_k^\#$ in the prefix, called a *failure state*, is further partitioned by refining the equivalence classes in such a way that the spurious counterexample is removed. To refine $s_k^\#$, the algorithm classifies the concrete states $s \in s_k^\#$ in three classes:

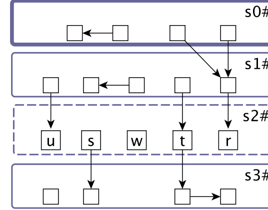


Fig. 2. Dead, bad and irrelevant states

- *Dead states:* they are reachable states $s \in s_k^\#$ along the spurious counterexample prefix but they have no outgoing transitions to the next states in the spurious counterexample, i.e., there is some concrete path prefix $\pi \in \gamma(\{s_i^\#\}_{i\in[0,k]})$ such that $s \in \pi$ and for any $s' \in s_{k+1}^\#$ it holds $\neg R(s, s')$.
- *Bad states:* they are non-reachable states $s \in s_k^\#$ along the spurious counterexample prefix but have outgoing transitions that cause the spurious counterexample, i.e., for any concrete path prefix $\pi \in \gamma(\{s_i^\#\}_{i\in[0,k]})$ we have $s \notin \pi$, but $R(s, s')$ for some concrete state $s' \in s_{k+1}^\#$.
- *Irrelevant states:* they are neither dead nor bad, i.e., they are not reachable and have no outgoing transitions to the next states in the counterexample.

Example 1 (Dead, bad and irrelevant states). Consider the abstract path prefix $\{s_0^\#, s_1^\#, s_2^\#, s_3^\#\}$ in Fig. 2. Each abstract state is represented as a set of concrete states (the smaller squares). The arrows are the transitions of the concrete Kripke structure and they induce abstract arcs in the obvious way. We use a thicker borderline to mark $s_0^\#$ as an initial abstract state and a dashed borderline to mark $s_2^\#$ as a failure state. The only dead state in $s_2^\#$ is r, because it can be reached via a concrete path from an initial state but there is no outgoing transition to a state in $s_3^\#$. The states s and t are bad, because they are not reachable from initial states, but have outgoing transitions to states in $s_3^\#$. The states u and w are irrelevant.

CEGAR looks for the coarsest partition that separates bad states from dead states. The partition is obtained by refining the partition associated with each variable. The chosen refinement is not local to the failure state, but it applies globally to all states: It defines a new abstract Kripke structure for which the spurious counterexample is no longer valid. Finding the coarsest partition corresponds to keeping the size of the new abstract Kripke structure as small as possible. This is known to be a NP-hard problem [2], due to the combinatorial

explosion in the number of ways in which irrelevant states can be combined with dead or bad states. In practice, CEGAR applies a heuristic: irrelevant states are not combined with dead states. The opposite option of separating bad states from both dead and irrelevant states is also viable.

In the following we assume that states in Σ are defined as assignments of values to a finite set of variables $x_1, ..., x_n$ that can take values in finite domains $D_1, ..., D_n$. Partitions over states are induced by partitions on domains. A *partition* P of variables $x_1, ..., x_n$ is a function that sends each x_i to a partition $P_i = P(x_i) \subseteq \wp(D_i)$. Given the abstractions associated with partitions $P_1, ..., P_n$ of the domains $D_1, ..., D_n$, the states of the abstract Kripke structure are defined by the possible ranges of values that are assigned to each variable according to the corresponding partition.

2.3 Programs

We let $\mathcal{P}$ be the set of programs written in the syntax of guarded commands [5] (e.g., in the style of CSP, Occam, XC), according to the grammar below:

$$d ::= x \in D \mid d, d \qquad g ::= x \in V \mid true \mid g \wedge g \mid g \vee g \mid \neg g$$
$$a ::= x = e \mid a, a \qquad c ::= g \Rightarrow a \mid c|c \qquad p ::= (d; g; c)$$

where x is a variable, $V \subseteq \bigcup_i D_i$ is a finite set of values, and e is a well-defined expression over variables. A *declaration* is a non-empty list of basic declarations $x \in D$ assigning a domain D to the variable x. We assume that all the variables appearing in a declaration d are distinct. A *basic guard* is a membership predicate $x \in V$ or true. A *guard* g is a formula of propositional logic over basic guards. We write $x \notin V$ for $\neg(x \in V)$. An *action* is a non-empty list of assignments. A single assignment $x = e$ evaluates the expression e in the current state and updates x accordingly. If multiple assignments $x_1 = e_1, ..., x_k = e_k$ are present, the expressions $e_1, ..., e_k$ are evaluated in the current state and their values are assigned to the respective variables. All the variables appearing in the left-hand side of multiple assignments must be distinct, so the order of assignments is not relevant. A *basic command* consists of a guarded command $g \Rightarrow a$: it checks if the guard g is satisfied by the current state, in which case it executes the action a to update the state. Commands can be composed in parallel: any guarded command whose guard is satisfied by the current state can be applied. A *program* $(d; g; c)$ consists of a declaration d, an initialisation proposition g and a command c, where all the variables in g and c are declared in d.

Example 2 (A sample program). We consider the following running example program (in pseudocode) that computes in y the square of the initial value of x.

```
1: y = 0;
2: while (x>0) {
3:     y = y + 2*x - 1;
4:     x = x - 1;
5: } output(y);
```

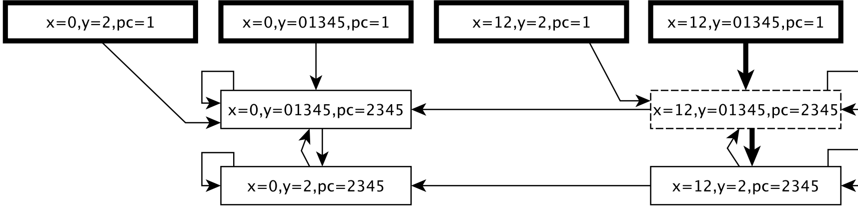


Fig. 3. An abstract Kripke structure

For simplicity we assume the possible values assigned to variables are in quite limited ranges, but starting with larger sets of values would not change the outcome of the application of the CEGAR algorithm. The translation of the previous program in the syntax of guarded commands is the program $p = (d\ ;\ g\ ;\ c_1|c_{2a}|c_{2b}|c_3|c_4)$, written below in CSP-like syntax. Intuitively, it is obtained by adding an explicit variable pc for the program-counter and then encoding each line of the source code as a basic command.

```
def x in {0,1,2} , y in {0,1,2,3,4,5} , pc in {1,2,3,4,5};  % d
init pc = 1;                                                 % g
do pc in {1}              => pc=2, y=0                       % c1  \
[] pc in {2} /\ x notin {0} => pc=3                          % c2a |
[] pc in {2} /\ x in {0}  => pc=5                            % c2b > c
[] pc in {3}              => pc=4, y=y+(2*x)-1               % c3  |
[] pc in {4}              => pc=2, x=x-1                     % c4  /
od
```

In this context, an attacker may want to check if y is ever assigned the value 2, which can be expressed as the property: $\phi \stackrel{\text{def}}{=} \forall\, G\ (pc \in \{1\} \vee y \notin \{2\})$ (i.e. for all paths, for all states in the path it is never the case that $pc \neq 1$ and $y = 2$).

Let $d = (x_1 \in D_1, ..., x_n \in D_n)$. A state $s = (x_1 = v_1, ..., x_n = v_n)$ of the program $(d; g; c)$ is an assignment of values to all variables in d, such that for all $i \in [1, n]$ we have $v_i \in D_i$ and we write $s(x)$ for the value assigned to x in s. Given $c = (g_1 \Rightarrow a_1| \cdots |g_k \Rightarrow a_k)$, we write $s \models g_j$ if the guard g_j holds in s and $s[a_j]$ for the state obtained by updating s with the assignment a_j.

The concrete Kripke structure $\mathcal{K}(p) = \langle \Sigma, R, I, \|\cdot\| \rangle$ associated with $p = (d; g; c)$ is defined as follows: the set of states Σ is the set of all states of the program; the set of transitions R is the set of all and only arcs (s, s') such that there is a guarded command $g_j \Rightarrow a_j$ in c with $s \models g_j$ and $s' = s[a_j]$; the set of initial states I is the set of all and only states that satisfy the guard g; the set of propositions is the set of all sentences of the form $x_i \in V$ where $i \in [1, n]$ and $V \subseteq D_i$; the interpretation function is such that $\| x_i \in V \| = \{s \mid s(x_i) \in V\}$.

Example 3 (A step of CEGAR). The Kripke structure associated with the program p from Example 2 has 90 states, one for each possible combination of values for its variables x, y, pc. There are 18 initial states: those where $pc = 1$.

Assume that the attacker, in order to prove ϕ, starts with the following initial partition (see [5]):

$$x : \{\{0\}, \{1, 2\}\} \qquad y : \{\{2\}, \{0, 1, 3, 4, 5\}\} \qquad pc : \{\{1\}, \{2, 3, 4, 5\}\} \quad (1)$$

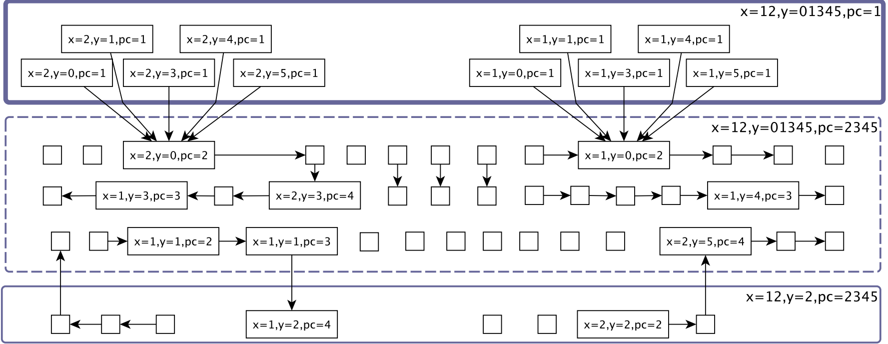


Fig. 4. Failure state

The corresponding abstract Kripke structure has just 8 states (see Fig. 3) with 4 initial states marked with bold borderline in Fig. 3, where, to improve readability, we write, e.g., $y = 01345$ instead of the more verbose $y \in \{0, 1, 3, 4, 5\}$.

There are several paths that lead to counterexamples for ϕ. One such path is the one marked with bold arrows in Fig. 3. It is detailed in Fig. 4 by showing the underlying concrete states. It is a spurious counterexample, because there is no underlying concrete path. The abstract failure state is $(x \in \{1, 2\}, y \in \{0, 1, 3, 4, 5\}, pc \in \{2, 3, 4, 5\})$, depicted with dashed borderline in Fig. 3. It contains one bad concrete state $(x = 1, y = 1, pc = 3)$, two dead states $((x = 1, y = 0, pc = 2)$ and $(x = 2, y = 0, pc = 2))$ and 37 irrelevant states.

By partition refinement we get the following refined partition:

$$x : \{\{0\}, \{1, 2\}\} \qquad y : \{\{2\}, \{0\}, \{1, 3, 4, 5\}\} \qquad pc : \{\{1\}, \{2\}, \{3, 4, 5\}\} \quad (2)$$

Thus the corresponding abstract Kripke structure has now 18 states, six of which are initial states. While the previously considered spurious counterexample has been removed, another one can be found and, therefore, CEGAR must be repeated, (see Fig. 6 discussed in Example 5 for further steps).

3 Model Deformations

We introduce a systematic model deformation making abstract model checking harder. The idea is to transform the Kripke structure, by adding states and transitions in a conservative way. We want to somehow preserve the semantics of the model, while making the abstract model checking less efficient, in the sense that only trivial (identical) partitions can be used to prove the property. In other words, any non-trivial abstraction induces at least one spurious counterexample.

Let $\mathbb{M}$ be the domain of all models specified as Kripke structures. Formally, a *model deformation* is a mapping between Kripke structures $\mathfrak{D} : \mathbb{M} \to \mathbb{M}$ such that for a given formula ϕ and $\mathcal{K} \in \mathbb{M}$: $\mathcal{K} \models \phi \;\Rightarrow\; \mathfrak{D}(\mathcal{K}) \models \phi$ and there exists a partition P such that $\mathcal{K}_P \models \phi \;\Rightarrow\; \mathfrak{D}(\mathcal{K}_P) \not\models \phi$. In this case we say that $\mathfrak{D}$

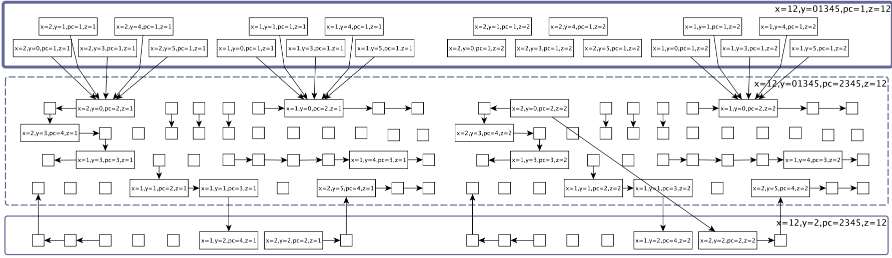


Fig. 5. A detail of a deformation

is a deformation for the partition P. Thus, a model deformation makes abstract model checking imprecise yet keeping the validity of the given formula.

In Section 4, we shall show that the deformation of the Kripke structures we consider are induced by transformations of the source program, that act as an obfuscation strategy against an attack specified by an abstract model-checker. Accordingly, we say that an *obfuscation* is a transformation $\mathcal{O} : \mathcal{P} \to \mathcal{P}$ such that for a given formula ϕ and program $p \in \mathcal{P}$: $\mathcal{K}(p) \models \phi \;\Rightarrow\; \mathcal{K}(\mathcal{O}(p)) \models \phi$ and there exists a partition P such that $\mathcal{K}(p)_P \models \phi \;\Rightarrow\; \mathcal{K}(\mathcal{O}(p))_P \not\models \phi$.

Example 4 (Model deformation). Consider the program p from Example 2. The first step of refinement with the CEGAR algorithm with the initial partition (1) (described in Example 3) results in the partition (2). Intuitively, a deformation of the Kripke structure that forced the CEGAR algorithm to split the sets of variable values in classes smaller than the ones in partition (2) would weaken the power of CEGAR. To this aim, consider a deformation $\mathfrak{D}(\mathcal{K})$ of the concrete Kripke structure $\mathcal{K}$ of Example 3 obtained by duplicating $\mathcal{K}$ in such a way that one copy is kept isomorphic to the original one, while the second copy is modified by adding and removing some transitions to make the CEGAR algorithm less efficient. The copies can be obtained by introducing a new variable $z \in \{1, 2\}$: for $z = 1$ we preserve all transitions, while for $z = 2$ we change them to force a finer partition when a step of the CEGAR algorithm is applied. For example, in the replica for $z = 2$, let us transform the copy of the dead state $(x = 2, y = 0, pc = 2)$ into a bad state. This is obtained by adding and removing some transitions. After this transformation, assuming an initial partition analogous to partition (1),

$$x : \{\{0\}, \{1, 2\}\} \quad y : \{\{2\}, \{0, 1, 3, 4, 5\}\} \quad pc : \{\{1\}, \{2, 3, 4, 5\}\} \quad z : \{\{1, 2\}\}$$

where all the values of the new variable z are kept together, we obtain an abstract Kripke structure isomorphic to the one of Fig. 3, with the same counterexamples. However, when we focus on the spurious counterexample, the situation is slightly changed. This is shown in Fig. 5, where the relevant point is the overall shape of the model and not the actual identity of each node. Roughly it combines two copies of the states in Fig. 4: those with $z = 1$ are on the left and those with $z = 2$ are on the right. The abstract state

$$(x \in \{1, 2\}, y \in \{0, 1, 3, 4, 5\}, pc \in \{2, 3, 4, 5\}, z \in \{1, 2\})$$

is still a failure state, but it has three bad states and three dead states:

bad states	dead states
$(x = 1 , y = 1 , pc = 3 , z = 1)$	$(x = 1 , y = 0 , pc = 2 , z = 1)$
$(x = 1 , y = 1 , pc = 3 , z = 2)$	$(x = 1 , y = 0 , pc = 2 , z = 2)$
$(x = 2 , y = 0 , pc = 2 , z = 2)$	$(x = 2 , y = 0 , pc = 2 , z = 1)$

The bad state $(x = 2, y = 0, pc = 2, z = 2)$ and the dead states $(x = 1, y = 0, pc = 2, z = 2)$ and $(x = 1, y = 4, pc = 3, z = 1)$ are incompatible. Therefore, the refinement leads to the partition below, where all values of x are separated:

$$x : \{\{0\}, \{1\}, \{2\}\}\ z : \{\{1\}, \{2\}\}\ y : \{\{2\}, \{0\}, \{1, 3, 4, 5\}\}\ pc : \{\{1\}, \{2\}, \{3, 4, 5\}\}$$

3.1 Measuring Obfuscations

Intuitively, the largest is the size of the abstract Kripke structure to be model checked without spurious counterexamples, the harder is for the attacker to reach its goal. The interesting case is of course when the property ϕ holds true, but the abstraction used by the attacker leads to spurious counterexamples.

We propose to measure and compare obfuscations on the basis of the size of the final abstract Kripke structure where the property can be directly proved. As the abstract states are generated by a partition of the domains of each variable, the size is obtained just as the product of the number of partition classes for each variable. As obfuscations can introduce any number of additional variables over arbitrary domains, we consider only the size induced by the variables in the original program (otherwise increasing the number of variables could increase the measure of obfuscation without necessarily making CEGAR ineffective).

In the following we assume that p is a program with variables X and variables of interest $Y \subseteq X$ and ϕ is the formula that the attacker wants to prove.

Definition 1 (Size of a partition). *Given a partition P of X, we define the* size *of P w.r.t. Y as the natural number $\prod_{y \in Y} |P(y)|$.*

Definition 2 (Measure of obfuscation). *Let $\mathcal{O}(p)$ be an obfuscated program. The* measure *of $\mathcal{O}(p)$ w.r.t. ϕ and Y, written $\#^Y_\phi \mathcal{O}(p)$, is the size of the final partition P w.r.t. Y as computed by the above model of the attacker.*

Our definition is parametric w.r.t. to the heuristics implemented in *check* (choice of the counterexample) and *refine* (how to partition irrelevant states). There are two main reasons for which the above measure is more significant than other choices, like counting the number of refinements: i) it is independent from the order in which spurious counterexamples are eliminated; ii) since the attacker has limited resources, a good obfuscation is the one that forces the attacker to model check a Kripke structure as large as possible.

Definition 3 (Comparing obfuscations). *The obfuscated program $\mathcal{O}_1(p)$ is as good as $\mathcal{O}_2(p)$, written $\mathcal{O}_1(p) \geq^Y_\phi \mathcal{O}_2(p)$, if $\#^Y_\phi \mathcal{O}_1(p) \geq \#^Y_\phi \mathcal{O}_2(p)$.*

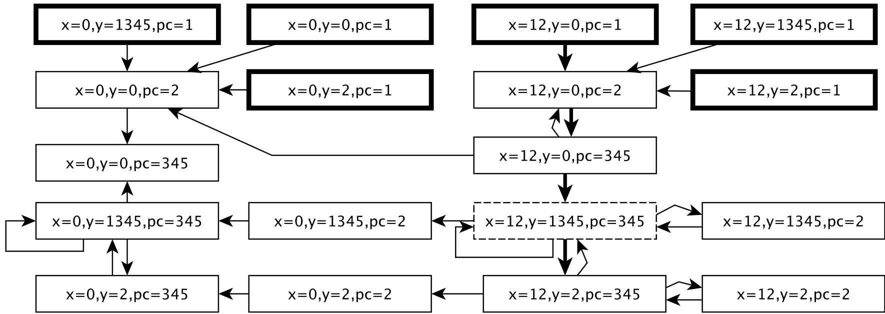


Fig. 6. A refined abstract Kripke structure

It follows that the best measure associated with an obfuscation is $\prod_{y \in Y} |D_y|$, where D_y denotes the domain of y. This is the case where the abstraction separates all the concrete values that the variables in Y may assume.

Example 5 (Ctd.). Let us consider again the running example and compare it with the semantically equivalent program p' below:

```
def x in {0,1,2},  y in {0,1,2,3,4,5}, pc in {1,2};
init pc = 1;
do pc in {1} => pc=2, y=x*x
od
```

Let x, y be the variables of interest. We have $\#_{\phi}^{\{x,y\}} p' = 2$, because the initial partition (1) is sufficient to prove that the property ϕ holds.

For the obfuscated program p, the size of the initial partition is just 4 (see partition (1) and the corresponding abstract Kripke structure in Fig. 3), and after one step of the CEGAR refinement the size of the computed partition is 6 (see partition (2) and the corresponding Kripke structure in Fig. 6). Since spurious counterexamples are still present, one more step of refinement is needed. When the attacker executes the procedure on the failure state marked with dashed border in Fig. 6, the result is the partition

$$x : \{\{0\}, \{1\}, \{2\}\} \qquad y : \{\{0\}, \{1\}, \{2\}, \{3\}, \{4, 5\}\} \qquad pc : \{\{1\}, \{2\}, \{4\}, \{3, 5\}\}$$

whose size is 15 as it has been necessary to split all values for x and most values for y. Now no more (spurious) counterexample can be found because all the states that invalidate the property are not reachable from initial states. Thus $\#_{\phi}^{\{x,y\}} p = 15$, while the best obfuscation would have measure 18, which is the product of the sizes of the domains of x and y. As the reader may expect, we conclude that $p \geq_{\phi}^{\{x,y\}} p'$. In our running example, for simplicity, we have exploited a very small domain for each variable, but if we take larger domains of values, then $\#_{\phi}^{\{x,y\}} p'$ and $\#_{\phi}^{\{x,y\}} p$ remain unchanged, while the measure of the best possible obfuscation would grow considerably.

4 Best Code Obfuscation Strategy

In the following, given an abstract Kripke structure K and a property ϕ, we let S_ϕ denote the set of abstract states that contain only concrete states that satisfy ϕ, and $S_{\overline{\phi}}$ be the set with at least one concrete state that does not satisfy ϕ. We denote by $\overline{\mathcal{O}}_\phi$ the obfuscation strategy realized by the following algorithm.

```
Input: program  p, property φ
 1: P = init(p, φ);
 2: K = kripke(P, p);
 3: (p, K, w, v_w) = fresh(p, K);
 4: (p, K, z, v_z) = fresh(p, K);
 5: S = cover(P);
 6: foreach s# ∈ S {
 // failure path preparation (lines 7-12, Cases 1-2)
 7:     π# = failurepath(s#, K, p, φ);
 8:     while (π# == null) {
 9:         if (not reach(s#, K, p, φ)
10:               (p, K, v_w) = makereachable(s#, K, p, φ, w, v_w);
11:         else (p, K, v_w) = makefailstate(s#, K, p, φ, w, v_w);
12:         π# = failurepath(s#, K, p, φ); }
//  main cycle (lines 13-19, Case 3)
13:     foreach (x_i, v_1, v_2) ∈ compatible(s#, π#, p) {
14:         (s, t) = pick(x_i, v_1, v_2, s#);
15:         if dead(t, s#, π#, p)
16:               (p, K, v_z) = dead2bad(t, s#, π#, K, p, z, v_z);
17:         else if bad(t, s#, π#, p)
18:               (p, K, v_z) = bad2dead(t, s#, π#, K, p, z, v_z);
19:         else (p, K, v_z) = irr2dead(t, s#, π#, K, p, z, v_z); }
20: } return p;
```

The algorithm starts by computing an initial partition P and the corresponding abstract Kripke structure K. We want to modify the concrete Kripke structure so that CEGAR will split the abstract states in trivial partition classes for the variables of interest. The idea is to create several replicas of the concrete Kripke structure, such that one copy is preserved while the others will be changed by introducing and deleting arcs. This is obtained by introducing a new variable z over a suitable domain $D_z = \{1, ..., n\}$ such that the concrete Kripke structure is replicated for each value z can take. As a matter of notation, we denote by $(s, z = v)$ the copy of the concrete state s where $z = v$. Without loss of generality, we assume that for $z = 1$ we keep the original concrete Kripke structure. In practice such value of z is hidden by an opaque expression. Actually we use two fresh variables, named w and z (lines 3 and 4): the former is used to introduce spurious counterexamples and failure states in the replica and the latter to force the splitting of failure states into trivial partition classes. The function *fresh* updates the program p and the Kripke structure K by taking into account the new variables and initializes the variables v_w and v_z that keep track of the last used values for w and z. When a new replica is created, such values are incremented.

The function *cover* (at line 5) takes the initial partition P and returns a set of abstract states $s_1^\#, ..., s_k^\#$, called a *covering*, such that, together, they cover all non-trivial[4] partition classes of the domains of the variable of interest, i.e. for each variable x_i, with $i \in [1, n]$, for each class $C \in P(x_i)$ such that $|C| > 1$ there is an abstract state $s_j^\#$ with $j \in [1, k]$ and a concrete state $s \in s_j^\#$ such that $s(x_i) \in C$. Note that the set of all abstract states is a (redundant) covering.

For each $s^\# \in \{s_1^\#, ..., s_k^\#\}$ in the covering, there are three possibilities:

1. $s^\#$ does not contain any concrete state that is reachable via a concrete path that traverses only abstract states in S_ϕ;
2. $s^\#$ is not a failure state but it contains at least one concrete state that is reachable via a concrete path that traverses only abstract states in S_ϕ;
3. $s^\#$ is already the failure state of a spurious counterexample.

In case (3), *failurepath*$(s^\#, K, p, \phi)$ (line 7) returns an abstract path that is a counterexample for ϕ, in the other cases the function *failurepath*$(s^\#, K, p, \phi)$ returns *null* and the algorithm enters a cycle (to be executed at most twice, see lines 8–12) that transforms the Kripke structure and the program to move from cases (1–2) back to case (3), in a way that we will explain later.

Case (3) (lines 13–19). The core of the algorithm applies to a failure state $s^\#$ of a spurious counterexample $\pi^\#$. In this case the obfuscation method will force CEGAR to split the failure state $s^\#$ by separating all values in the domains of the variables of interest. Remember that CEGAR classifies the concrete states in $s^\#$ in three classes (bad, dead and irrelevant) and that dead states cannot be merged with bad or irrelevant states. We say that two states that can be merged are *compatible*. The role of the new copies of the Kripke structure is to prevent any merge between concrete states in the set $s^\#$. This is done by making sure that whenever two concrete states $(s, z = 1), (t, z = 1) \in s^\#$ can be merged into the same partition, then the states $(s, z = v_z + 1)$ and $(t, z = v_z + 1)$ cannot be merged together, because one is dead and the other is bad or irrelevant.

The function *compatible*$(s^\#, \pi^\#, p)$ returns the set of triples $(x_i, \{v_1, v_2\})$ such that x_i is a variable of interest and any pair of states $(s, x_i = v_1), (s, x_i = v_2) \in s^\#$ that differ just for the value of x_i are compatible. Thus, the cycle at line 13 considers all such triples to make them incompatible. At line 14, we pick any two compatible states $(s, z = 1)$ and $(t, z = 1)$ such that $s(x_i) = v_1$, $t(x_i) = v_2$ and $s(x) = t(x)$ for any variable $x \neq x_i$. Given the spurious counterexample $\pi^\#$ with failure state $s^\#$ and a concrete state $t \in s^\#$ for the program p, the predicate *dead*$(\cdot)$ returns true if the state $(t, z = 1)$ is dead (line 15). Similarly, the predicate *bad*$(\cdot)$ returns true if the state $(t, z = 1)$ is bad (line 17).

If $(t, z = 1)$ is dead (w.r.t. $s^\#$ and $\pi^\#$), then it means that $(s, z = 1)$ is also dead (because they are compatible), so we apply a dead-to-bad transformation to t in the replica for $z = v_z + 1$. This is achieved by invoking the function *dead2bad*$(\cdot)$ (line 16) to be described below. The transformations *bad2dead*$(\cdot)$ (line 18) and *irr2dead*$(\cdot)$ (line 19) apply to the other classifications for $(t, z = 1)$.

[4] A partition class is trivial if it contains only one value.

In the following, given a concrete state $s = (x_1 = w_1, ..., x_n = w_n)$, we denote by $G(s)$ the guard $x_1 \in \{w_1\} \wedge ... \wedge x_n \in \{w_n\}$ and by $A(s)$ the assignment $x_1 = w_1, ..., x_n = w_n$. Without loss of generality, we assume for brevity that the abstract counterexample is formed by the abstract path prefix $\pi^\# = \{s_0^\#, s_1^\#, s^\#, s_2^\#\}$, with $s^\#$ the failure state and t is the concrete state in $s^\#$ that we want to transform (see Figs. 7–9).

dead2bad$(\cdot)$. To make t bad, the function must remove all concrete paths to t along the abstract counterexample $\pi^\#$ and add one arc from t to some concrete state t' in $s_2^\#$. To remove all concrete paths it is enough to remove the arcs from states in $s_1^\#$ to t (see Fig. 7). At the code level, *dead2bad*$(\cdot)$ modifies each command $g \Rightarrow a$ such that there is some $s' \in s_1^\#$ with $s' \models g$ and $t = s'[a]$. Given t and $s_1^\#$, let $S(g \Rightarrow a) = \{s' \in s_1^\# \mid s' \models g \wedge t = s[a]\}$. Each command $c = (g \Rightarrow a)$ such that $S(c) \neq \emptyset$ is changed to the command

$$g \wedge \left(z \notin \{v_z + 1\} \vee \bigwedge_{s' \in S(c)} \neg G(s')\right) \Rightarrow a.$$

When $z \neq v_z + 1$ the updated command is applicable whenever c was applicable. When $z = v_z + 1$ the command is not applicable to the states in $S(c)$. To add the arc from t to a state t' in $s_2^\#$ it is enough to add the command $z \in \{v_z + 1\} \wedge G(t) \Rightarrow A(t')$.

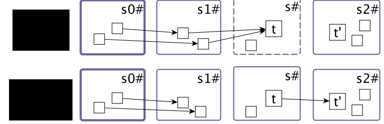


Fig. 7. From dead to bad

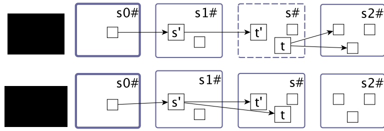


Fig. 8. From bad to dead

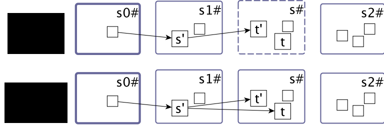


Fig. 9. From irrelevant to dead

bad2dead$(\cdot)$. The function selects a dead state t' in the failure state $s^\#$ and a concrete path $\pi = \langle s_0, s', t' \rangle$ to t' along the abstract counterexample $\pi^\#$. To make t dead in the replica with $z = v_z + 1$, the function adds a concrete arc from s' to t and removes all arcs leaving from t to concrete states in $s_2^\#$ (see Fig. 8). To insert the arc from s' to t, the command $G(s') \wedge z \in \{v_z + 1\} \Rightarrow A(t)$ is added. To remove all arcs leaving from t to concrete states in $s_2^\#$, the function changes the guard g of each command $g \Rightarrow a$ such that $t \models g$ and $t[a] \in s_2^\#$ to $g \wedge (z \notin \{v_z + 1\} \vee \neg G(t))$, which is applicable to all the states different from t where $g \Rightarrow a$ was applicable as well as to the replicas of t for $z \neq v_z + 1$.

irr2dead$(\cdot)$. Here the state t is irrelevant and we want to make it dead. The function builds a concrete path to t along the abstract counterexample $\pi^\#$. As before, it selects a dead state t' in the failure state and a concrete path $\pi = \langle s_0, s', t' \rangle$ to s along the abstract counterexample $\langle s_0^\#, s_1^\#, s^\# \rangle$. To make t dead the function adds an arc from s' (i.e., the state that immediately precedes t' in π) to t (see Fig. 9). For the program it is sufficient to add a new command with guard $G(s') \wedge z \in \{v_z + 1\}$ and whose assignment is $A(t)$.

From cases (1–2) to case (3) (lines 7–12). The predicate $reach(s^{\#}, K, p, \phi)$ is true if we are in case (2) and false if we are in case (1). In both cases we apply some preliminary transformations to K and p after which $s^{\#}$ is brought to case (3). The function $makereachable(s^{\#}, K, p, \phi, w, v_w)$ transforms the Kripke structure so that in the end $s^{\#}$ contains at least one concrete state that is reachable via a concrete path that traverses only abstract states in S_{ϕ}, moving from case (1) to cases (2) or (3), while the function $makefailstate(s^{\#}, K, p, \phi, w, v_w)$ takes $s^{\#}$ satisfying case (2) and it returns a modified Kripke structure where $s^{\#}$ now falls in case (3). The deformations are illustrated in Figs. 10–11. At the code level, addition and removal of arcs is realized as detailed before.

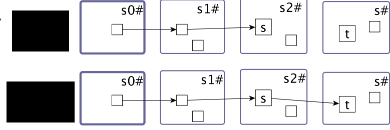


Fig. 10. Case 1: $makereachable(\cdot)$

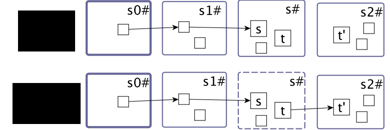


Fig. 11. Case 2: $makefailstate(\cdot)$

5 Main Results

Our obfuscation preserves the semantics of the program. This is because all the transformations we have discussed maintain the original Kripke structure associated with a distinguished value of the new variables w and z that are introduced. Indeed,when the obfuscated program is executed with initial values $w = 1$ and $z = 1$ it behaves exactly as the original program. By exploiting opaque expressions to initialise the variables w and z, we hide their values from the attacker who has to take into account all possible values for w and z and thus run CEGAR on the deformated Kripke structure.

Theorem 1 (Embedding). *Let* $p = (d; g; c)$ *and* $\overline{\mathcal{O}}_{\phi}(p) = ((d, w \in D_w, z \in D_z); g; c')$*, then* $\mathcal{K}(p)$ *is isomorphic to* $\mathcal{K}((d, w \in \{1\}, z \in \{1\}); g; c')$*.*

The isomorphism at the level of Kripke structures guarantees that the obfuscation does not affect the number of steps required by any computation, i.e., to some extent the efficiency of the original program is also preserved.

Second, the obfuscation preserves the property ϕ of interest when the program is executed with any input data for w and z, i.e. ϕ is valid in all replicas.

Theorem 2 (Soundness). $\mathcal{K}(p) \models \phi$ *iff* $\mathcal{K}(\overline{\mathcal{O}}_{\phi}(p)) \models \phi$*.*

Note that Theorem 1 guarantees that the semantics is preserved entirely, i.e. not only ϕ is preserved in all replicas (Theorem 2) but any other property is preserved when the obfuscated program is run with $w \in \{1\}$ and $z \in \{1\}$.

The next result guarantees the optimality of obfuscated programs.

Theorem 3 (Hardness). $\#^{Y}_{\phi}\overline{\mathcal{O}}_{\phi}(p) = \prod_{y \in Y} |D_y|$.

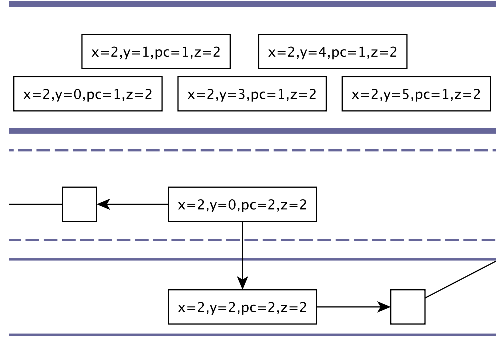


Fig. 12. A fragment of the model for $z = 2$

As a consequence, for any program p and formula ϕ the function mapping $\mathcal{K}(p)$ into $\mathcal{K}(\overline{\mathcal{O}}_\phi(p))$ is a model deformation for all (non-trivial) partitions of the variables of interest.

Theorem 4 (Complexity). *The complexity of our best code obfuscation strategy is polynomial in the size of the domains of the variables of interest Y.*

Example 6 (Best code obfuscation). Consider the abstract Kripke structure in Fig. 3. The failure state $s^\# = (x \in \{1, 2\}, y \in \{0, 1, 3, 4, 5\}, pc \in \{2, 3, 4, 5\})$ covers all the non-trivial partition classes for the variables of interest x and y. Since it is a failure state for an abstract counterexample, we are in case (3). For simplicity, since the transformations for cases (1–2) are not needed, we omit the insertion of the variable w.

The dead state $(x = 1, y = 0, pc = 2)$ is incompatible with the irrelevant state $(x = 1, y = 1, pc = 2)$, thus the triple $(y, \{1, 2\})$ is incompatible. For the same reason the value 0 for y is also separated from the values $3, 4, 5$. Our obfuscation must separate the values $1, 2$ for x and the values $1, 3, 4, 5$ for y. Therefore at most 6 replicas are needed. In the end, 5 values for z suffices. Let us take the triple $(x, \{1, 2\})$ and let us pick the two dead states $t = (x = 2, y = 0, pc = 2)$ and $s = (x = 1, y = 0, pc = 2)$ in Fig. 4. The algorithm invokes $\mathit{bad2dead}(\cdot)$ on state $(t, z = 2)$ to make it incompatible with the dead state $(x = 1, y = 0, pc = 2, z = 2)$. At the code level, we note that all incoming arcs of t are due to the command c_1 (see Fig. 4). To remove them, c_1 becomes $c_{1,1}$.

```
pc in {1} /\ (z notin {2} \/ x notin {2} \/ y notin {2}) => pc=2, y=0     %c11
```

Moreover, to make $(t, z = 2)$ a bad state, is added an arc from the state $(t, z = 2)$, to $(x = 2, y = 2, pc = 2, z = 2)$ with the new command $c_{1,2}$

```
pc in {1} /\ z in {3} /\ x in {1} /\ y in {1}            => pc=3          %c12
```

In Fig. 12 we show the relevant changes on the Kripke structure for the replica with $z = 2$ (compare it with Fig. 4). To complete the obfuscation more transformations are required: one bad-to-dead and two irrelevant-to-dead transformations. Finally, we obtain the program $po = \overline{\mathcal{O}}_\phi(p)$ below:

```
def x in {0,1,2} , y in {0,1,2,3,4,5} , pc in {1,2,3,4,5} , z in {1,2,3,4,5};
init pc = 1;
do pc in {1} /\(z notin {2} \/ x notin {2} \/ y notin {2})=> pc=2, y=0         %c11
[] pc in {1} /\ z in {3} /\ x in {1} /\ y in {1}          => pc=3              %c12
[] pc in {1} /\ z in {4} /\ x in {1} /\ y in {4}          => pc=3              %c13
[] pc in {1} /\ z in {5} /\ x in {2} /\ y in {5}          => pc=4              %c14
[] pc in {2} /\ x notin {0}                               => pc=3              %c2a
[] pc in {2} /\ x in {0}                                  => pc=5              %c2b
[] pc in {2} /\ x in {2}} /\ y in {0} /\ z in {2}         =>  y=2              %c21
[] pc in {3} /\(z notin {3} \/ x notin {1} \/ y notin {1})=> pc=4, y=y+(2*x)-1 %c31
[] pc in {4}                                              => pc=2, x=x-1       %c4
od
```

Let us assume that the attacker starts with the abstraction of the obfuscated program induced by the partition $x : \{\{0\}, \{1, 2\}\}$, $y : \{\{2\}, \{0, 1, 3, 4, 5\}\}$, $pc : \{\{1\}, \{2, 3, 4, 5\}\}$, and $z : \{\{1, 2, 3, 4, 5\}\}$. The abstract Kripke structure is isomorphic to the one in Fig. 3 having several spurious counterexamples for ϕ. One such path is similar to the one in Fig. 3: $\{s_0^\#, s_1^\#, s_2^\#\}$ with:

$$
\begin{aligned}
s_0^\# &= x \in \{1,2\}, y \in \{0,1,3,4,5\}, pc \in \{1\}, z \in \{1,2,3,4,5\} \\
s_1^\# &= x \in \{1,2\}, y \in \{0,1,3,4,5\}, pc \in \{2,3,4,5\}, z \in \{1,2,3,4,5\} \\
s_2^\# &= x \in \{1,2\}, y \in \{2\}, pc \in \{2,3,4,5\}, z \in \{1,2,3,4,5\}.
\end{aligned}
$$

The failure state is $s_1^\#$. It has 5 bad concrete states and 12 dead states. By CEGAR we get the partition that has only trivial (singletons) classes. Therefore the abstract Kripke structure coincides with the concrete Kripke structure: it has 450 states of which 90 are initial states.

Given that the variables of interest are x and y, the measure of the obfuscation is 18, i.e., it has the maximum value and thus $po \geq_\phi^{\{x,y\}} p \geq_\phi^{\{x,y\}} p'$. We remark that when $z = 1$, po has the same semantics as p and p'.

The guarded command po can be understood as a low-level, flattened description for programs written in any language. However, it is not difficult to derive, e.g., an ordinary imperative program from a given guarded command. We do so for the reader's convenience.

```
 1:      z = opaque1(x,y,z);
 2: pc1: if ( ( z!=2 || x!=2 || y!=2 ) && opaque2(x,y,z) ) { y=0; goto pc2; }
 3:      else if ( z=3 && x =1 && y=1 ) goto pc3;
 4:      else if ( z=4 && x =1 && y=4 ) goto pc3;
 5:      else if ( z=5 && x =2 && y=5 ) goto pc4;
 6: pc2: if ( x=2 && y=0 && z=2 && opaque3(x,y,z) ) y=2;
 7:      while ( x>0 ) {
 8: pc3:    if ( z!=3 || x!=1 || y!=1 ) y = y + 2*x - 1;
 9: pc4:    x = x - 1;
10: pc5: } output(y);
```

To hide the real value of z we initialise the variable using an opaque expression $opaque1(x, y, z)$ whose value is 1. Moreover, one has to pay attention to the possible sources of nondeterminism, which can arise when there are two or more guarded commands $g_1 \Rightarrow a_1$ and $g_2 \Rightarrow a_2$ and a state s such that $s \models g_1$ and $s \models g_2$. The idea is to introduce opaque predicates so that the exact conditions under which a branch is taken are hard to determine by the attacker, who has to take into account both possibility (*true* and *false*) as a nondeterministic choice.

In our example, the sources of nondeterminism are due to the pairs of commands $(c_{1,1}, c_{1,2})$, $(c_{1,1}, c_{1,3})$, $(c_{1,1}, c_{1,4})$ and $(c_{2,1}, c_{2a})$. Consequently, we assume two opaque predicates $opaque2(x, y, z)$ and $opaque3(x, y, z)$ are available. In order to preserve the semantics, for $z = 1$ we require that $opaque1(x, y, z)$ returns *true*, while $opaque2(x, y, z)$ is unconstrained. Finally, since the program counter is an explicit variable in guarded commands, we represent its possible values by labels and use goto instructions accordingly. Thus we write the label *pcn* to denote states where $pc = n$ and write **goto** *pcn* for assignments of the form $pc = n$.

6 Discussion

We have shown that it is possible to systematically transform Kripke structures in order to make automated abstraction refinement by CEGAR hard. Addressing refinement procedures instead of specific abstractions makes our approach independent from the chosen abstraction in the attack.

To enforce the protection of the real values of variable w (and analogously for z) initialized by opaque functions against more powerful attacks able to inspect the memory of different program runs, one idea is to use a class of values instead of a single value. This allows the obfuscated code to introduce instructions that assign to w different values in the same class, thus convincing the attacker that the value of w is not invariant.

The complexity of our best code obfuscation strategy is polynomial in the size of the domains of the variables of interest. Moreover, we note that the same algorithm can produce a valuable obfuscation even if one selects a partial cover instead of a complete one: in this case, it is still guaranteed that the refinement strategy will be forced to split all the values appearing in the partial cover. This allows to choose the right trade-off between the complexity of the obfuscation strategy and the measure of the obfuscated program.

As already mentioned, our obfuscation assumes that CEGAR makes dead states incompatible with both irrelevant and bad states. Our algorithm can be generalised to the more general setting where dead states are only incompatible with bad states. Therefore even if the attacker had the power to compute the coarsest partition that separates bad states from dead states (which is a NP-hard problem) our strategy would force the partition to consist of trivial classes only.

We can see abstraction refinement as a learning procedure which learns the coarsest state equivalence by model checking a temporal formula. Our results provide a very first attempt to defeat this procedure.

As an ongoing work, we have extended our approach to address attacks aimed to disclose data-flow properties and watermarks. It remains to be investigated how big the text of the best obfuscated program can grow: limiting its size is especially important in the case of embedded systems.

We plan to extend our approach to other abstraction refinements, like predicate refinement and the completeness refinement in [16] for generic abstract interpreters and more in general for a machine learning algorithm. This would make automated reverse engineering hard in more general attack models.

Acknowledgement. We are very grateful to Alberto Lluch-Lafuente for the fruitful discussions we had on the subject of this paper.

References

1. S. Banescu, C. S. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In S. Schwab, W. K. Robertson, and D. Balzarotti, editors, *Proc. 32nd Annual Conference on Computer Security Applications, ACSAC 2016*, pages 189–200. ACM, 2016.
2. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
3. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc. of the 19th ACM Symp. on Principles of Programming Languages (POPL '92)*, pages 343–354. ACM Press, 1992.
4. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
5. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
6. C. Collberg, J. Davidson, R. Giacobazzi, Y. Gu, A. Herzberg, and F. Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 26(6):8–13, 2011.
7. C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.
9. P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Proc. of the 31st ACM Symp. on Principles of Programming Languages (POPL '04)*, pages 173–185. ACM Press, New York, NY, 2004.
10. M. Dalla Preda and R. Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.
11. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
12. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass., 1990.
13. R. Giacobazzi. Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In *Proc. of the 6th IEEE Int. Conferences on Software Engineering and Formal Methods (SEFM '08)*, pages 7–20. IEEE Press, 2008.
14. R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*, pages 63–72. ACM Press, 2012.
15. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In *Proc. of the 8th Int. Static Analysis Symp. (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2001.
16. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *Journal of the ACM*, 47(2):361–416, March 2000.

17. Microsoft. Static driver verifier website (last consulted november 2017), 2017. `https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier`.
18. J. Nagra, C. D. Thomborson, and C. Collberg. A functional taxonomy for software watermarking. *Aust. Comput. Sci. Commun.*, 24(1):177–186, 2002.
19. F. Ranzato and F. Tapparo. Generalized strong preservation by abstract interpretation. *Journal of Logic and Computation*, 17(1):157–197, 2007.
20. D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In D. B. MacQueen and L. Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 38–48. ACM, 1998.
21. D. A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In G. Levi, editor, *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 1998.
22. R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *Proc. 4th Int. Workshop on Information Hiding (IHW '01)*, volume 2137 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 2001.
23. C. Wang, J. Hill, J. C. Knight, and J. W. Davidson. Protection of software-based survivability mechanisms. In *2001 International Conference on Dependable Systems and Networks (DSN 2001) (formerly: FTCS), 1-4 July 2001, Göteborg, Sweden, Proceedings*, pages 193–202. IEEE Computer Society, 2001.

Abstract Code Injection

A Semantic Approach Based on Abstract Non-Interference

Samuele Buro and Isabella Mastroeni

Department of Computer Science, University of Verona
Strada le Grazie 15, 37134 Verona, Italy
{samuele.buro,isabella.mastroeni}@univr.it

Abstract. Code injection attacks have been the most critical security risks for almost a decade. These attacks are due to an *interference* between an *untrusted input* (potentially controlled by an attacker) and the execution of a *string-to-code* statement, interpreting as code its parameter. In this paper, we provide a semantic-based model for code injection parametric on what the programmer considers *safe behaviors*. In particular, we provide a *general* (abstract) non-interference-based framework for *abstract code injection* policies, i.e., policies characterizing safety against code injection w.r.t. a given specification of safe behaviors. We expect the new semantic perspective on code injection to provide a deeper knowledge on the nature itself of this security threat. Moreover, we devise a mechanism for enforcing (abstract) code injection policies, *soundly* detecting attacks, i.e., avoiding false negatives.

1 Introduction

Security is an enabling technology, hence security means power. The correct functionality and coordination of large scale organizations, e-government, web services, in general, relies on confidentiality and integrity of data exchanged between different agents, and on the proper functioning of the applications. These features, almost unavoidable, become real opportunities for the attackers seeking to disclose and/or corrupt valuable information or, more widely, to break security.

According to OWASP (Open Web Application Security Project) [1], the most critical security risks have been application level injections attacks for almost a decade [21,22,23]. The reason of their success and their spread is twofold: An *easy exploitability* of vulnerabilities and a *severe impact* of attacks. In other words, code injection bugs allow attackers to cause extensive damage for minimum effort. Despite this, organizations often underestimate their consequences, and the inevitable result has been a recent history full of this kind of attacks [29].

Several approaches [3,10,11,18,19,24,28,30,31], have been studied for *preventing* code injection, but only few focus on the harder problem of *defining* it [3,24,28]. Indeed, the intuition of what can be classified as an injection attack is quite straightforward, and it is clearly provided in the following informal definition [23]:

> "Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. Attackers trick the interpreter into executing unintended

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 116–137, 2018.
https://doi.org/10.1007/978-3-319-73721-8_6

```php
<?php

$id = $argv[0];
$query  = "SELECT * FROM users WHERE id = $id;";
$result = pg_query($conn, $query);

?>
```

```jsp
<html><body>
<%
   String user =
      request.getParameter("user");
%>
<h1>Welcome <%= user %></h1>
</body></html>
```

Fig. 1: Example of SQLi in a PHP program and of XSS in a JSP page.

> commands via supplying specially crafted data. Injection flaws allow attackers to create, read, update, or delete any arbitrary data available [...]."

Unfortunately, this intuitive definition does not help in formalizing a general definition of code injection since there is a clear problem in formalizing the concept of *unintended commands*. Moreover, this notion may depend on several factors, e.g., the kind of application, the environment of execution, etc.

The essence of code injection. Code injection is a wide category of attacks where an attacker exploits the presence of an *untrusted input* (i.e., an input whose source is potentially untrusted) for injecting code (*unintended commands*) that will affect the execution of a *string-to-code* statement (which interprets as code its parameter), altering the output behavior of the application.

The main types of attacks in this category are undoubtedly SQL injection (SQLi) and cross-site scripting (XSS) attacks. In SQLi the attacker attempts to execute an arbitrary query on a database server. An example is given on the left in Fig. 1 where the attacker is able to extract the whole content of the users table by injecting the value `"3 OR 1 = 1"` in the untrusted input `argv[0]`, making the `query` condition a tautology. In XSS the attacker attempts to execute a client-side code (e.g., JavaScript) in the user's browser. In the program on the right in Fig. 1, an attacker can execute his own JavaScript code in the victim's browser by injecting, for example, the string `"<script>alert('message')</script>"`. Other kinds of code injection attacks such as command injection, `eval`-injection, XPath injection, remote file injection, etc., also play a central role in this category.

Despite the multi-faceted nature of code injection, all these attacks present a key common feature: *A code executed by an interpreter which is* dependent on *the value of an untrusted input, that alters the intended semantics of the application, making the execution unsafe.*

Up to date solutions for facing code injection. The prevention techniques against code injection are a well-studied topic of applications security, and they can be classified into two categories: The techniques that follow an industrial/technical approach and those based on formal methods.

In the former case, applications are made secure by validating inputs (escaping, whitelists, blacklists, etc., of values), by parametrizing queries (i.e., by separating

the parameters binding from the code compilation — mainly used in the SQLi context) and/or by using other ad hoc mechanisms. Even though they provide some degree of security and robustness to programs, they suffer from well-known flaws [2,10,13,18,30]. Moreover, they do not aid the programmers through the process of securing applications.

In the latter case, there are several formal approaches claimed to be *sound* and/or *complete* w.r.t. a given notion of code injection [28,31,10,24]. The majority of them are mainly focused on the SQLi attacks than on the broader problem of code injection, and they rely on dynamic taint analysis algorithms [11,19,24,31] or parsing trees [4,28] in order to detect alterations of the syntactic queries structure on the basis of fixed policies. To the best of our knowledge, the works that mainly rely on the problem of *defining* code injection are [3,24,28]. All of them provide a *syntactic-based* notion of code injection and the two most related to our approach are [3,24]:

- In [3], the core idea is to dynamically mine the programmer-intended query structure on any input, and to detect attacks by comparing them against the intended query structure;
- In [24], an application's output is considered a code injection attack if there exists at least one tainted symbol which is *code*, i.e., it is not a fully evaluated value.

It is worth noting that these definitions are indeed specific instances of the informal definition given in the introduction. In particular, both fix a precise notion of what a programmer could consider as *unintended commands*. This loss of generality is clearly useful in practice, since it provides a decidable and easy way to detect potential code injection attacks, but it may reduce flexibility, since the programmer might need to weaken or strengthen the fixed notion, depending of the environment of execution of developed applications.

Our solution: A semantic-based approach. In this paper, we propose to shift these syntactic notions towards a semantic model of code injection — as suggested in [24] — in order to broaden the generality of the definition.

The key point of the whole approach we propose is based on a simple observation: Each time an expression e is *executed* in a string-to-code statement (e.g., in a query execution, in an `eval` statement, . . .), there is a set of states[1] such that the execution of e in one of these states leads the program to an unintended/unsafe state. Since we are focusing on code injection, we can restrict this set only to those states *depending* on at least one injected (untrusted) value. In other words, we have injection whenever there is a program statement whose parameters *depend on* an untrusted input and whose execution causes an *unsafe output behavior*, namely the attacker can lead the program execution to show unsafe behaviors. For instance, in the code on the left of Fig. 1, the query execution statement `pg_query` depends on the untrusted input `argv[0]`.

[1] Intuitively, think of a state as all the information concerning the program execution at each step of computation.

This kind of data dependency is precisely what is called *interference* in language-based security [9,26]. This means that, safety against code injection can be seen as a non-interference policy. Being more precise, it is clear that we do not care to model any possible interference, since any *dynamic* code is expected to depend on the input in some way. There is a potential security breach only when the dependency causes a variation between what is considered safe and what is considered unsafe. In this sense, the right non-interference framework to consider is *abstract non-interference* [8], where the interference between *properties* of inputs and *properties* of outputs is studied. Hence, we define *abstract code injection policies* parametric on what the programmer considers safe output behaviors.

It is clear that, if we could provide a universal characterization of what is a *safe output behavior* (holding for all programs, for all execution environments), we could design a tool enforcing (abstract) code injection policies for any program.

Unfortunately, in real settings, different programs, or even the same program in different execution contexts, may require different instances of the policy. Consider a music streaming web application P_{music} with premium and free users. The first ones have access to both copyright and copyright-free music, while the second ones can only listen to copyright-free songs. In order to encourage free users to buy premium subscriptions, the programmers allow them to add a copyrighted song in their music library once per month, chosen from a list of top five hits. Suppose the user chooses the first song and its code number (e.g., `83`) is submitted to the web application as a GET parameter: `https://webappmusic.com/load_library.php?choice=83`. To load user's library, the web application executes the following query where the variable `$free_codes_list` contains the codes of all the copyright-free songs and the variable `$user_choice` contains the code `83`[2]:

```
SELECT * FROM songs WHERE code IN ($free_codes_list, $user_choice)
                                   ↓
SELECT * FROM songs WHERE code IN (5, 3, 2, 54, 32, 21, 12, ..., 83)
```

If the GET parameter `choice` is not validated, an evil user can inject an arbitrarily long list of values, loading more than one song in his/her library, for instance

```
SELECT * FROM songs WHERE code IN (5, 3, 2, 54, 32, 21, 12, ..., 83, 43, 23, ...)
```

Consider now a web application P_{doc} allowing users to download documents from a list of `pdf` files. An user provides the documents' codes he/she wishes to download (e.g., `2, 23, 6`) and an HTTP request is sent to the web application: `https://webappdoc.com/download.php?doc[]=2&doc[]=23&doc=6`. Suppose the user's choice is stored in the PHP variable `$doc_list` and the following query is executed:

```
SELECT * FROM docs WHERE code IN ($doc_list)
                   ↓
SELECT * FROM docs WHERE code IN (2, 23, 6)
```

[2] The highlighted code is the injected one.

In this case, a list of values has been injected but, contrarily to the P_{music} scenario, it is not to be considered as an attack, since the programmer's intention and the context are different. It follows that every model fixing a notion of *unintended commands* will provide a wrong answer to, at least, one of the two examples. Even worse, let us change the first example by supposing that the programmers decide to allow the user to choose two songs: A list of two songs now has not to be considered an attack. These trivial examples show how, even in the same context, the programmer's intention, and therefore what is unintended, may vary.

It is worth noting that a common feature in these examples is that, for controlling code injection we have to partition inputs into two subsets: The set of inputs producing safe output behaviors after the query execution, and all the others, generating unsafe behaviors. With these considerations in mind, the model we propose is based on the following key points:

1. Abstract code injection policies can be defined in terms of an output characterization of safe output behaviors, potentially determined by the programmer;
2. If the program does not satisfy this policy, it means that there are values of untrusted inputs able to change the output (observable) behavior of the program, making it unsafe. We call *safe inputs* those always leading to safe output behaviors;
3. The abstract non-interference framework [8] allows us to characterize the partition of inputs leading to different (safe/unsafe) output behaviors. This suggests us what should be verified on the input of the application.

At this point, in order to control code injection vulnerabilities we propose to go through two phases: First, we characterize the abstract code injection policy to enforce, for instance by asking the programmer to specify safe inputs and/or safe output behaviors; Second, we enforce the chosen policy. The latter phase could be tackled both statically, by manually patching the program (but in this case we lose flexibility), or by monitoring the program, i.e., by dynamically checking whether the executed inputs are safe, w.r.t. some decidable characterization.

Hence, we propose a static analysis for aiding the programmer to understand when a safe input specification is necessary, and consequently asking the programmer to annotate the program with information characterizing the abstract injection policy to enforce. Then we propose the design of a dynamic analysis, i.e., a monitor checking whether the execution violate the abstract injection policy.

2 Background

The core language WhileFun. In order to show our approach, we define the core language WhileFun that encloses all the important features from the code injection point of view. WhileFun is dynamically typed, based on a classic While language augmented with functions. A valid WhileFun program (denoted by $P \in$ WhileFun) consists in a `main` function (the entry point, non-callable by the code) and eventual user-defined functions. We assume that only a subset of the parameters of the `main` function may be *untrusted inputs*. Furthermore, we introduce the syntactic category *str2code* of string-to-code statements:

```
str2code ::= exec(exp)     SQL query execution       (SQL injection)
           | eval(exp)     Code execution            (eval-injection)
           | system(exp)   Command/Shell execution   (Command injection)
           | show(exp)     Webpage displaying        (XSS)
```

All these commands send the evaluated expression *exp* (a string of code) to the corresponding interpreter. For the sake of simplicity, we assume that *str2code* commands are only allowed in the `main` function. The language syntax and the semantics of the other commands are standard.

Program semantics. $\mathcal{Vars}$ denotes the set of program and environment variables[3], and $\mathcal{Val}$ the set of values. $\mathcal{L}$ denotes the set of *line numbers* (program points). Let $l \in \mathcal{L}$, and $Stm(l)$ be the statement at program line l. For a given program P, we denote by $\mathcal{L}_P \subseteq \mathcal{L}$ the set of all and only the line numbers corresponding to statements of the program P, i.e., $\mathcal{L}_P = \{ l \in \mathcal{L} \mid Stm(l) \in P \}$.

A *program state* $\sigma \in \mathcal{S}$ is a pair $\langle n^k, \mu \rangle$ where n is the executed program point, k is the number of times the statement $Stm(n)$ has been reached so far (in the following we will call n^k *execution point*), μ is the memory [17]. A *memory* $\mu \in \mathcal{Mem}$ is a map $\mu : \mathcal{Vars} \to \mathcal{Val}$ mapping variables to values such that $\mu(x)$ is the value of x in μ, while $\mu[x \leftarrow v]$ is the memory μ' such that $\forall y \neq x. \mu'(y) = \mu(y)$, while $\mu(x) = v$. For simplicity, we denote by $\mathcal{Val}^x$ the set of values over which x can range, i.e., the domain of x. Furthermore, we define the equivalence relation $=_x$ between two memories μ and μ': $\mu =_x \mu' \iff \forall y \neq x.\ \mu(y) = \mu'(y)$.

A *state trajectory* $\tau \in \mathcal{T} = \mathcal{S}^* \cup \mathcal{S}^\omega$ is a sequence of program states through which a program goes during the execution. Any initial state has $n^k = 1^1$, i.e., the set of initial states is $\mathcal{S}_\iota = \{ \langle 1^1, \mu \rangle \mid \mu \in \mathcal{Mem} \}$. The state trajectory obtained by executing program P from the input memory μ is denoted by $\langle\!| P |\!\rangle(\langle 1^1, \mu \rangle)$ and $\langle\!| P |\!\rangle^{n^k}(\langle 1^1, \mu \rangle)$ is the prefix of $\langle\!| P |\!\rangle(\langle 1^1, \mu \rangle)$ whose last state has execution point n^k. The denotational semantics of $P \in$ WhileFun is the function $[\![P]\!] : \mathcal{S} \to \mathcal{S}$ providing the I/O characterization of program semantics. Let $\langle 1^1, \mu_\iota \rangle \in \mathcal{S}_\iota$, the denotational semantics is defined as $[\![P]\!](\langle 1^1, \mu_\iota \rangle) = \sigma_\dashv$ where $\sigma_\dashv$ is the last state of $\langle\!| P |\!\rangle(\langle 1^1, \mu_\iota \rangle)$ if it is finite, $\perp$ otherwise [6]. We similarly define $[\![P]\!]^{n^k}(\langle 1^1, \mu_\iota \rangle)$, the denotational semantics w.r.t. to the *execution point* n^k.

Static Single Assignment (SSA) SSA [7] is a well known code representation where the def-use chains are made explicit. This is an intermediate non-executable representation of code, used by compilers for simplifying some static analyses. In the SSA form, each assignment generates a new unique name (usually denoted by a numerical subscript) for the defined variable, and all the uses reached by that definition are renamed. An example is shown in Fig. 2a where the program on the left is rewritten in the one on the right. If different definitions reach the

[3] Without loss of generality, we assume that the state of the *str2code* interpreter (for instance, the state of the database when the interpreter is the database server) is modeled in the set of variables $\mathcal{Vars}$. This means that the memory μ contains all the observable information concerning both the program and environment.

```
V := 4         V1 := 4
Z := V + 5     Z1 := V1 + 5
V := 6         V2 := 6
W := V + 7     W1 := V2 + 7
```

(a) Linear SSA transformation.

```
if (P)                       if (P)
  then { V := 4 }              then { V1 := 4 }
  else { V := 6 }              else { V2 := 6 }
                             V3 := φ(V1, V2)
```

(b) SSA transformation with ϕ-function.

Fig. 2: Examples of SSA program representations [7].

same use of a given identifier, a special form of assignment, called *ϕ-function*, is added: This is a special assignment identifying the join of several definitions of the same identifier (an example is given in Fig. 2b). The presence of these ϕ-functions makes the code not-executable but there exist standard techniques for reconstructing executable programs from the SSA form: By replacing the ϕ-functions with assignment operations, and by dropping subscripts [7].

Reaching definitions analysis (RD). Reaching definitions analysis (RD for short), determines the definitions potentially reaching each use of an identifier. In a control flow graph (CFG), a definition reaches a node if there is a path from the definition to the node, along which the defined variable is never redefined. On the SSA form this analysis becomes trivial since the reaching definition is precisely the unique definition of the used identifier (see [20] for details).

3 Defining Abstract Code Injection

In this section, we define the notion of *abstract code injection* policy, which consists in a code injection policy *parametric on* the programmer characterization of safe/unsafe output behaviors. More specifically, a code injection vulnerability is a potential *interference* between an untrusted input and the execution of a string-to-code statement. We say that a program does not suffer of a code injection vulnerability if it *enforces a code injection policy*, meaning that any code injection vulnerability in the program is avoided.

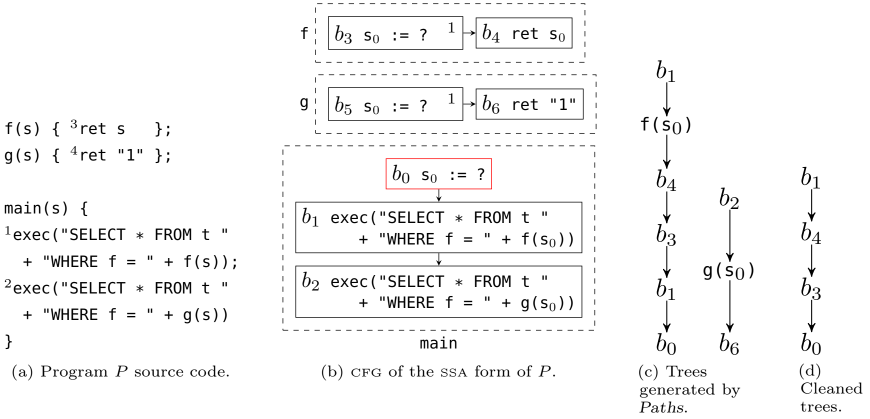
First of all, let us define formally code injection policies in terms of non-interference. In particular, we define a notion of non-interference between an input and a program point, e.g., the program point of the string-to-code statement. We recall that $[\![P]\!]^{n^k}$ (see Sect. 2) computes the state at the execution point n^k.

Definition 1 ($\mathrm{NI}^{\mathtt{x}}_{\mathtt{P}}(n)$). *Let $P \in$ WHILEFUN be a program, x be an input of P and $n \in \mathcal{L}_P$. We say that x is* non-interfering *at the program point n of P iff*

$$\forall k \in \mathbb{N}\,.\ \mathrm{NI}^{x}_{P}(n,k)$$

where, for any $k \in \mathbb{N}$, $\mathrm{NI}^{x}_{P}(n,k)$ (x non-interfering *at the execution point n^k in P) holds iff*

$$\mu_0 =_x \mu'_0 \implies [\![P]\!]^{n^k}(\langle 1^1, \mu_0\rangle) = [\![P]\!]^{n^k}(\langle 1^1, \mu'_0\rangle)$$



(a) Program P source code.

(b) CFG of the SSA form of P.

(c) Trees generated by *Paths*.

(d) Cleaned trees.

Fig. 3: Steps of static analysis algorithms.

Intuitively, this notion states that, during the execution of P, whenever the execution reaches the program point n, even if we change the initial value of x, the observable behavior of P does not change. It is self-evident that an attacker cannot perform a successfully injection attack on x if the above definition holds for all the program points where a string-to-code statement is executed.

For instance, consider the string-to-code statement `¹exec("SELECT * FROM t WHERE f = " + f(s))` in Fig. 3a. There exist two values $v_1 = \mathtt{1}, v_2 = \mathtt{2}$ generating two different queries (observable behaviors) after its execution. On the other hand, if we consider the statement `²exec("SELECT * FROM t WHERE f = " + g(s))`, then for each $v_1, v_2 \in \mathcal{V}al^{\mathtt{s}}$ we have that $[\![P]\!]^{2^1}(\langle 1^1, \{\,\mathtt{s} \to v_1\,\}\rangle) = [\![P]\!]^{2^1}(\langle 1^1, \{\,\mathtt{s} \to v_2\,\}\rangle)$ since `g(s)` is a constant function.

Exactly as it happens in language-based security, this notion of non-interference (and therefore of code injection policy) is in general too strong, since it does not allow us to really distinguish between safe and potentially unsafe code. In particular, this definition says that the only safe code is the one not depending on untrusted inputs, which is in general not acceptable: String-to-code statements, such as query executions, code evaluations, etc., *have* to be dependent on the user's input. For this reason, we need to formalize code injection parametrically on what the programmer considers a *safe (output) behavior* and/or which are the programmer *(expected) safe inputs*, leading only to safe outputs.

Formally, let $\mathbb{O} \subseteq \mathcal{M}em$ be the set of all the output states considered safe by the programmer after the execution of an string-to-code statement ($\mathbb{O}$ is the set of safe output behaviors). We can define the characteristic map of $\mathbb{O}$ as

$$\rho_{\mathbb{O}}(\langle n^k, \mu\rangle) = \begin{cases} \mathit{true} & \text{if } \mu \in \mathbb{O} \\ \mathit{false} & \text{otherwise} \end{cases}$$

At this point we can weaken Def. 1 defining abstract code injection policies, parametric on the programmers characterization of safe outputs $\mathbb{O}$ as done for abstract non-interference [8,14,15].

Definition 2 $(\text{ANI}^{\text{x}}_{\text{P}}(\mathbb{O}, n))$**.** *Let $P \in$* WhileFun *be a program, x be an input of P, $n \in \mathcal{L}_P$ and $\mathbb{O}$ be the set of the safe output behaviors. We say that x is* non-interfering *w.r.t. $\mathbb{O}$ at the program point n in P iff*

$$\forall k \in \mathbb{N} \,.\, \text{ANI}^{x}_{P}(\mathbb{O}, n, k)$$

where, for any $k \in \mathbb{N}$, $\text{ANI}^{x}_{P}(\mathbb{O}, n, k)$ (x is (abstract) non-interfering *w.r.t. $\mathbb{O}$ with the execution point n^k in P) iff*

$$\mu_0 =_x \mu_0' \implies \rho_{\mathbb{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0 \rangle)) = \rho_{\mathbb{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0' \rangle))$$

We say that a program enforces an *abstract code injection* policy, w.r.t. a safe output behaviors characterization $\mathbb{O}$, if does not exist any untrusted inputs x and string-to-code statements in P (at the program line n) such that $\neg\, \text{ANI}^{\text{x}}_{\text{P}}(\mathbb{O}, n)$.

The abstract non-interference framework [8,14,15] allows us to move further and to characterize an enforcing strategy when an abstract code injection policy is not satisfied. It should be clear that also $\text{ANI}^{\text{x}}_{\text{P}}(\mathbb{O}, n)$ is a too strong property, in the sense that most "raw" programs cannot satisfy it, unless they show, independently from the input, always the same kind of behavior (safe or unsafe).[4] For all the other programs, where attackers have the possibility of exploiting an untrusted input for leading to unsafe output behaviors, namely those *vulnerable* to code injection, the abstract non-interference framework allows us to characterize which variation of inputs causes the safe/unsafe variation of output behaviors. In other words, in this framework it is possible to determine the input binary partition for every input x (defined by the characterization function $\phi^x : \mathcal{V}al^x \to \{\mathit{true}, \mathit{false}\}$) making the following equation to hold for each $v_1, v_2 \in \mathcal{V}al^x$ [8,16]:

$$\phi^x(v_1) = \phi^x(v_2) \implies \rho_{\mathbb{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle) = \rho_{\mathbb{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle))$$

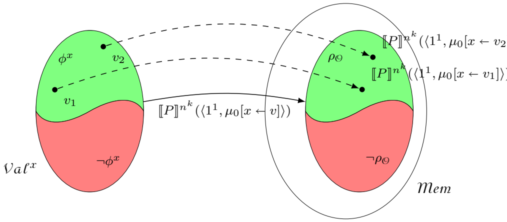


Fig. 4: The bipartions induced by $\mathbb{O}$.

Namely, for each pair of values for x, both in the same equivalence class of ϕ^x ($\phi^x(v_1) = \phi^x(v_2) = \mathit{true}$ or $\phi^x(v_1) = \phi^x(v_2) = \mathit{false}$) the output behavior is always respectively safe or unsafe. In Fig. 4 we depict the situation. All and only the values in $\mathcal{V}al^x$, satisfying ϕ^x leads the execution of P in n^k to satisfy $\rho_{\mathbb{O}}$.

We can formally characterize the partition ϕ^x *enforcing* an abstract code injection policy $\text{ANI}^{\text{x}}_{\text{P}}(\mathbb{O}, n)$ (simply denoted $\phi^x_{\mathbb{O}}$), as follows [8]:

$$\forall v \in \mathcal{V}al^x \,.\, (\phi^x_{\mathbb{O}}(v) \iff \rho_{\mathbb{O}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v] \rangle))) \tag{1}$$

[4] Note that, programs showing always unsafe behaviors are of no interest since they are unsafe by nature, the attacker cannot force unsafety.

It is worth noting that, the set $\mathcal{I}^x_{\mathbb{O}} = \{\ v \in \mathcal{Val}^x \mid \phi^x_{\mathbb{O}}(v) = \mathit{true}\ \}$ is precisely the *language of safe inputs w.r.t.* $\mathbb{O}$, i.e., those inputs leading to safe outputs.

Note that, we call $\phi^x_{\mathbb{O}}$ an *enforcing strategy*, since it characterizes what we should check in order to avoid code injection w.r.t. $\mathbb{O}$. Once we know which are the safe inputs, we could check whether the received inputs, during computation, are safe. The possibilities are two: The programmer could patch the code implementing all the checks, but this reduces the approach flexibility (if $\mathbb{O}$ changes then the code has to be partially rewritten); The programmer could augment the code with input annotations that can be checked dynamically. In the following section, we propose a monitor-based approach. The choice is driven by the idea of having a flexible enforcing technique, allowing the programmer to change $\mathbb{O}$ without changing the code, but only the input annotations. The approach we propose, allows us a further degree of flexibility, allowing us to fix input annotations depending also on the dynamic execution path.

4 Enforcing Abstract Code Injection

In this section, we propose a technique for enforcing code injection policies w.r.t. the programmer characterization of safe output behaviors, namely able to recognize and stop executions potentially under a code injection attack.

In the previous section, we showed that starting from a characterization of safe output behaviors $\mathbb{O}$, we can characterize the language of safe inputs w.r.t. $\mathbb{O}$, i.e., $\mathcal{I}^x_{\mathbb{O}}$ for each input x. Unfortunately, even if $\mathbb{O}$ is decidable, the definition of $\mathcal{I}^x_{\mathbb{O}}$ does not guarantee in general its decidability, hence we propose a technique where the programmer provides a decidable language $\mathcal{I}^x$ of *acceptable* inputs *consistent* with $\mathbb{O}$, namely such that it satisfies the following inclusion

$$\forall v \in \mathcal{Val}^x . (v \in \mathcal{I}^x \ \Rightarrow\ \rho_{\mathbb{O}}([\![P]\!]^{n^k}(\langle 1^1, \mu_0[x \leftarrow v]\rangle)))$$

meaning that $\mathcal{I}^x \subseteq \mathcal{I}^x_{\mathbb{O}}$. This inclusion guarantees soundness, in the sense that it avoids false negatives, while it can admit false positives/alarms, since there are safe inputs that are not in the acceptable input language.

In general, it should be clear, that both soundness and completeness of the model w.r.t. real potential attack situations depend on the choice of $\mathbb{O}$ and of $\mathcal{I}^x$. An over-approximation of $\mathcal{I}^x$ or $\mathbb{O}$ (meaning that there are inputs or behaviors erroneously labeled as safe) may lead to false negatives, missing some attack situations and therefore losing soundness w.r.t. real potential attacks. An under-approximation of $\mathcal{I}^x$ or $\mathbb{O}$ (meaning that there are safe inputs or behaviors labeled as unsafe) may allow false positives/alarms, hence losing precision/completeness of the approach w.r.t. real potential attacks. In the following, we will always talk of soundness and completeness of the enforcing technique w.r.t. the chosen model, i.e., w.r.t. the choice of all $\mathcal{I}^x$ and/or $\mathbb{O}$.

4.1 A contract-based approach for enforcing abstract code injection

The approach we propose is based on the idea of asking the programmer the language $\mathcal{I}^x$ of acceptable input values for each untrusted input. In order to

avoid useless contracts for untrusted inputs not leading to the execution of a string-to-code statement, we aid the programmer providing him/her both the inputs and the paths potentially vulnerable to code injection, and therefore requiring him/her to fix a corresponding language of safe inputs $\mathcal{I}^x$, that we call *contract*. Then we propose to annotate the code with these contracts in order to dynamically monitor the program execution for checking contracts only when necessary, namely when a string-to-code statement, depending on an untrusted input, is executed. The approach we propose is composed by three phases:

- *Static analysis:* The code is statically analyzed in order to extract the vulnerable paths, where some untrusted inputs interfere with the execution of a string-to-code statement, and therefore where the programmer should establish some input restrictions.
- *Contracts request:* Then we ask the programmer the contracts for all the *vulnerable paths*, i.e., those paths where an untrusted inputs reaches a string-to-code statements. This phase could be made automatic by providing a general/unique definition of restrictions, independently from the particular untrusted input and/or executed path. The result of this phase is an *annotated program* with contracts.
- *Monitor:* Finally, the monitor is able to kill all the executions of an annotated program, when a vulnerable path is executed and the involved untrusted input violates its contract. In other words, the result of this phase is a *monitored program*, namely the monitor specialized on the annotated program [12].

Basic notations. In literature, there are many variants of the CFG construction [5]: We choose to build a *single block intraprocedural CFG*, in which we consider the parameters of the `main` function to be potentially *untrusted inputs* (depicted in **red** in Fig. 3b), while the parameters of the other functions are considered *formal parameters* (depicted with a numerical superscript in Fig. 3b). We define the notion of *sub-path* of a path $b_0 \dots b_m$ in a CFG as a sequence of blocks $p_0 \dots p_n$ such that $p_0 = b_0$, $p_n = b_m$, and for each $0 < i < n$ if $p_{i-1} = b_k$ then $\exists b_j, j > k$ such that $p_i = b_j$, intuitively, it is a path of the CFG where some intermediate blocks are missing. Given a CFG C, if there exists a path p in it such that p' is a sub-path of p, then we say that p' is sub-path of C.

In addition, we define the following domains: $\mathcal{Blocks}$ is the set of blocks in the CFG, $\mathcal{FunCalls}$ is the set of all function calls (including arguments) in the SSA form of the program, $\mathcal{Fun}$ is the set of the defined functions, $\mathcal{Args}$ is the set of arguments of the function calls, and here $\mathcal{Vars}$ is the set of program variables in the SSA form. We also use the following well known functions: USE to compute the variables used and the function calls performed in a statement or, by extension, in a block, RET to compute the set of the returning points of a given function, and RD to compute the RD analysis. We call *ground block* any block b such that $\text{USE}(b) = \varnothing$.

$\mathcal{Trees}$ is the set of trees whose nodes are either blocks, or calls, or $\bot_{k \in \mathbb{N}}$ values. Let n be a tree node and $\mathbf{T} \subseteq \mathcal{Trees}$, the tree constructor is $\mathit{Tree}(n, \mathbf{T}) = \langle n : \mathbf{T} \rangle$ which, starting from a set of several trees $\mathbf{T}$, builds one new tree with root n and

sub-trees those in $\mathbf{T}$, i.e., it adds an edge (n, m) in $\langle n : \mathbf{T}\rangle$ for each node m root of a tree in $\mathbf{T}$. $\langle n\rangle \equiv \langle n : \varnothing\rangle$. On a tree T, we define the function BRANCHES(T) that returns the set of all the paths p, from the root to a leaf, in T, and the function REVERSE(p) that changes the direction of edges in the path p. We abuse notation by calling REVERSE also its additive lift to sets of paths.

4.2 Static phase and contract request

The purpose of the static phase is to detect *where* the information flows within a program under analysis. In particular, we are interested in all the vulnerable paths, i.e., those starting from an untrusted input and affecting a string-to-code statement. We explain our approach and algorithms also by using as running example the code in Fig. 3a.

CFG and SSA construction. Let P be the program under analysis. The first two steps of the analysis consist in the construction of the program representation that will be used for performing the analysis. First, we build the control flow graph for each procedure f declared in the program (CFG(f)) obtaining the set $\mathbb{C} = \{\, \text{CFG}(f) \mid f \text{ procedure in } P\,\}$. After that, in order to improve the analysis (and, in particular, RD), we consider the SSA representation of each CFG (see Sect. 2 for details), where each variable is defined only once. Let SSA be the function that computes the SSA form of a given CFG. We define the set $\mathbb{C}^{ssa} = \{\, \text{SSA}(C) \mid C \in \mathbb{C}\,\}$ representing the SSA form of the program P (for an example, see Fig. 3b).

Trees construction. The trees construction is the core step of the static analysis: For each block b containing a string-to-code statement, the function *Paths* (Fig. 5) builds, backwards, the trees of potential execution paths, looking for the vulnerable ones.

Paths is a function with two parameters: The first one is either a block $b \in \mathcal{Blocks}$, or a procedure call $f(\mathbf{a}) \in \mathcal{FunCalls}$, or a special value $\bot_k$ $(k \in \mathbb{N})$; The second parameter is a *history* of function calls $\mathbf{c} = [c_1, \ldots, c_k]$ (ε denotes the empty history). A *function call* is a triple $(b, f, \mathbf{a})$ consisting in the calling block b, in the called function f, and in its sequence of actual parameters $\mathbf{a} = [a_1, \ldots, a_m]$. Given a block $b \in \mathcal{Blocks}$ containing a string-to-code statement and an initial empty sequence of calls $\mathbf{c} = \varepsilon$, the function *Paths* tracks backward the potential dependencies in order to identify which untrusted inputs may affect the string-to-code statement in b. These chains of dependencies form a tree having b as root and as leaves either a ground block, or bottom, or a block containing the interfering untrusted input (the latter case identifies vulnerable paths).

In order to formally define the function *Paths*, we have first to define the auxiliary function *Rec*, used for determining the arguments of recursive calls, i.e., whose aim is that of computing the parameters of the recursive step of *Paths*:

$$\begin{aligned} Rec\colon & (\mathcal{Blocks} \times (\mathcal{Vars} \cup \mathcal{FunCalls}) \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^*) \to \\ & (\mathcal{Blocks} \cup \mathcal{FunCalls} \cup \{\bot_k\}_{k\in\mathbb{N}}) \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \end{aligned}$$

$$Paths\colon (\mathcal{Blocks} \cup \mathcal{FunCalls} \cup \{\bot_k\}_{k\in\mathbb{N}}) \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$

$$Paths(arg, \mathbf{c}) = \begin{cases} Block(arg, \mathbf{c}) & \text{if } arg \in \mathcal{Blocks} \\ Call(arg, \mathbf{c}) & \text{if } arg \in \mathcal{FunCalls} \\ Bot(\bot_k, \mathbf{c}) & \text{otherwise} \end{cases}$$

where $\mathbf{c} = [c_1, \ldots, c_m]$ such that $\forall 1 \le i \le m \,.\, c_i = (b_i, f_i, \mathbf{a}^i)$ and $\mathbf{a}^i = [a^i_1, \ldots, a^i_{n_i}]$

$$Block\colon \mathcal{Blocks} \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$
$$Call\colon \mathcal{FunCalls} \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$
$$Bot\colon \{\bot_k\}_{k\in\mathbb{N}} \times (\mathcal{Blocks} \times \mathcal{Fun} \times \mathcal{Args})^* \to \mathcal{Trees}$$

$$\begin{cases}
(1) & Block(b, \mathbf{c}) & = Tree(b, \{\, Paths(Rec(b, u, \mathbf{c})) \mid u \in \text{USE}(b) \,\}) \quad \text{if USE}(b) \neq \varnothing \\
(2) & Block(b, \mathbf{c}) & = Tree(b, \varnothing) \quad \text{if USE}(b) = \varnothing \text{ and } b \text{ does not define a formal parameter} \\
(3) & Block(b, \mathbf{c}) & = Tree(b, \{\, Bot(\bot_k, \mathbf{c}) \,\}) \quad \text{if USE}(b) = \varnothing \text{ and } b \text{ defines the } k\text{-th formal parameter in } c_m \\
(4) & Call(f(\mathbf{a}), \mathbf{c}) & = Tree(f(\mathbf{a}), \{\, Paths(r, \mathbf{c}) \mid r \in \text{RET}(f) \,\}) \quad \text{if } \forall c_i, c_j (i \neq j) \in \mathbf{c} \,.\, f_i \neq f_j \\
(5) & Call(f(\mathbf{a}), \mathbf{c}) & = Tree(f(\mathbf{a}), \varnothing) \quad \text{if } \exists c_i, c_j (i \neq j) \in \mathbf{c} \,.\, f_i = f_j \\
(6) & Bot(\bot_k, \mathbf{c}) & = Tree(\bot, \varnothing) \quad \text{if USE}(a^m_k) = \varnothing \\
(7) & Bot(\bot_k, \mathbf{c}) & = Tree(b_m, \{\, Paths(Rec(b_m, u, [c_1, \ldots, c_{m-1}])) \mid u \in \text{USE}(a^m_k) \,\}) \quad \text{if USE}(a^m_k) \neq \varnothing
\end{cases}$$

Fig. 5: Definition of the function *Paths*.

$$Rec(b, arg, \mathbf{c}) = \begin{cases} (\text{RD}(b, v), \mathbf{c}) & \text{if } arg = v \in \mathcal{Vars} \\ (f(\mathbf{a}), [c_1, \ldots, c_m, (b, f, \mathbf{a})]) & \text{if } arg = f(\mathbf{a}) \in \mathcal{FunCalls} \end{cases}$$

Intuitively, if *arg* is a variable, *Rec* computes the RD analysis of *arg* from the block b looking for the variables which *arg* depends on, while if *arg* is a function call, *Rec* updates the history of function calls $\mathbf{c}$.

The tree computation algorithm starts calling $Paths(b, \varepsilon)$ for each block b containing a string-to-code statement. Let us explain the definition of $Paths(arg, \mathbf{c})$ given in Fig. 5 case by case:

$arg = b \in \mathcal{Blocks}$: $Paths(arg, \mathbf{c}) = Block(b, \mathbf{c})$

- USE$(b) \neq \varnothing$ (case (1)): If we reach a block b (not ground) containing one or more uses u, we create a tree with b as root, and as children all the trees resulting from calling *Paths* on all the recursive arguments (computed by $Rec(b, u, \mathbf{c})$, for each $u \in$ USE(b)).
- USE$(b) = \varnothing$ (cases (2) and (3)): When we reach a ground block b we have to distinguish two cases: When the block does not define any formal parameter (of the function including it), the analysis terminates on the current block b (case (2)); Otherwise, a subtree is created with root b and children the trees resulting from the analysis of the corresponding actual parameter by calling $Bot(\bot_k, \mathbf{c})$ (case (3)).

$arg = f(\mathbf{a}) \in \mathcal{FunCalls}$: $Paths(arg, \mathbf{c}) = Call(f(\mathbf{a}), \mathbf{c})$

When *Paths* is called on a function call $f(\mathbf{a})$, and the call has been already met before in $\mathbf{c}$, then the analysis stops adding $f(\mathbf{a})$ to the tree (case (5)). Otherwise, a subtree is created with $f(\mathbf{a})$ as root, and as children all the

trees resulting from calling *Paths* on all the return blocks ($\textsc{Ret}(f)$) of the function f (case (4)). The idea beyond this strategy comes from the fact that a function call can, in the worst case, only propagate, and not generate, flows.

$arg = \bot_k \in \{\bot_k\}_{k\in\mathbb{N}}$: $Paths(arg, \mathbf{c}) = Bot(\bot_k, \mathbf{c})$
We have $\bot_k$ when we reach the definition of a formal parameter (as explained in case (3)). If in the last performed call (m-th) the k-th formal parameter contains some uses ($\textsc{Use}(a_k^m) \neq \varnothing$) then we track back these uses calling *Paths* similarly to case (1) (case (7)). On the contrary, if $\textsc{Use}(a_k^m) = \varnothing$ the analysis stops adding $\bot_k$ to the tree (case (6)).

We define the set $\mathbb{T} = \{\, Paths(b_e, \varepsilon) \mid \exists C^{ssa} \in \mathbb{C}^{ssa} \,.\, b_e \in C^{ssa} \wedge b_e$ contains a string-to-code statement $\}$, i.e., the set of trees generated by the function *Paths*.

In the example in Fig. 3a, a code injection attack may be possible via the input s, since it interferes with the first query execution (as explained in Sect. 1), but not with second one. The tree generated by $Paths(b_1, \varepsilon)$ is depicted in Fig. 3c on the left. In this case the only leaf is the block b_0, in which an untrusted input s is required. This means that the corresponding path is a code injection vulnerable path. On the other hand, the resulting tree of the second query execution $Paths(b_2, \varepsilon)$ is given in Fig. 3c on the right. In this case, the path ends up in the block b_6 which is not an untrusted input, and therefore meaning that it is not a vulnerable path. Hence, in this example $\mathbb{T} = \{\langle b_1 : \langle f(s_0) : \langle b_4 : \langle b_3 : \langle b_1 : \langle b_0 \rangle\rangle\rangle\rangle\rangle\rangle, \langle b_2 : \langle g(s_0) : \langle b_6 \rangle\rangle\rangle\}$.

Once we have the set of trees $\mathbb{T}$ with all the paths leading from a string-to-code statement to either (i) a ground, or (ii) a bottom block or (iii) an untrusted input block, we discard all the safe paths, i.e., those not depending on untrusted inputs (cases (ii, iii)). In addition, we also need to remove non-executable blocks, i.e., those added during the algorithm computation and which do not correspond to application/code statements. These blocks are the ones related to function calls (added by *Paths* in cases (4) and (5)), which are part of the abstract syntax of others blocks and those re-added after the analysis of a function call (see case (7) in *Paths*). We define $\mathbb{T}^c$ as the set of all cleaned up trees in $\mathbb{T}$. In the running example, $\mathbb{T}^c = \{\langle b_1 : \langle b_4 : \langle b_3 : \langle b_0 \rangle\rangle\rangle\rangle\}$ (Fig. 3d). Finally, it is possible to make some transformations that will make easier to dynamically associate the executed path with the right contract. In particular, we reverse the paths in order to have them in the execution direction and we define the set $\mathbb{P} = \{\, \textsc{Reverse}(p) \mid \exists T^c \in \mathbb{T}^c$ such that $p \in \textsc{Branches}(T^c) \,\}$. In the example, $\mathbb{P} = \{b_0\, b_3\, b_4\, b_1\}$.

Contracts. At this point, the programmer, for each path in $\mathbb{P}$ leading to an untrusted input x has to provide a *contract*, i.e., a decidable language $\mathcal{I}^x$ (e.g., regular, context free, etc.) of acceptable values for that path:

$$\mathcal{C}ontracts = \{\, (\mathcal{I}^x, p) \mid \mathcal{I}^x \subseteq \mathcal{V}al^x \text{ decidable input language}, p \in \mathbb{P} \,\}$$

Hence, a contract $(\mathcal{I}^x, p)$ means that the value of the untrusted input in the first block of p, i.e., p_0, has to be in the language $\mathcal{I}^x$ if all the blocks in p have

Algorithm 1 *MonInt*

```
1: procedure MonInt((ℂ, Contracts)_P, μ0)
2:   V = ∅
3:   μ = μ0
4:   b = b0                      ▷ initial block of the program
5:   while b ≠ ⊥ do              ▷ ⊥ is the exit block
6:     D_Blocks = D_Blocks ∪ {b}
7:     if b ≠ b0 then D_Edge = D_Edge ∪ {(b_p, b)}
8:     if b contains an untrusted input x then
9:       V = V ∪ {(x, μ(x))}
10:    end if
11:    if b executes code then
12:      Verify(V, Contracts, D, b)
13:    end if
14:    b_p = b
15:    (b, μ) = INTERPRETER(b, μ, ℂ)
16:  end while
17: end procedure
```

Algorithm 2 *Verify*

```
1: procedure Verify(V, Contracts, D, b)
2:   for all c = (𝒥^x, p0 ··· pn) ∈ Contracts do
3:     if pn ≠ b then continue
4:     for i = 0 to n − 1 do
5:       if not REACHABILITY(D, p_i, p_{i+1}) then
6:         continue to the next contract c
7:       end if
8:     end for
9:     x = untrusted input variable in p0
10:    if V(x) ∉ 𝒥^x then Throw Exception
11:  end for
12: end procedure
```

Fig. 6: Monitor algorithm.

been executed. For instance, a contract $\mathcal{J}^{\mathtt{s}}$ for the reverse of path p of the tree in Fig. 3d could be expressed by the regular expression `0 | [1-9][0-9]`* to force the untrusted input `s` to be an integer.

Finally, given a program P to analyze, the static phase provides in output the pair $(\mathbb{C}, \mathcal{Contracts})_P$.

4.3 Dynamic phase

In this section, we explain how we intend to use the result of the static phase in order to provide a monitor, i.e., a dynamic checker, of potential code injection attacks. We observe that code injection is a safety property [27] since, once a string-to-code statement depends on an untrusted input at the program point of its execution, then a vulnerability definitively occurred, meaning that the only possibility for enforcing the safety property is to stop computation.

The monitor algorithm. In the following, we develop a monitor, exploiting the contracts verifier only when necessary. In particular, the idea is to design a monitor which executes directly the language interpreter on all the statements, except on string-to-code ones, for which the monitor has prior to check the satisfiability of (potentially many) contracts.

The *MonInt* procedure (Algorithm 1 in Fig. 6) takes as input the result $(\mathbb{C}, \mathcal{Contracts})_P$ of the static analysis on the program P and an initial memory μ_0. In order to determine the right contracts to check, the algorithm keeps up-to-date a dynamic structure D, in which the information of the path followed by the execution is stored: Every time a new block b is reached, the set of blocks is expanded by adding b into it (line 6), and a new edge from to previously executed block b_p to b is added to the set of edges (line 7); We will refer to

```
1  main(s) {
2    i := 0;
3    copy := "";
4    while (i <= length(s)) {
5      c := s[i];
6      if      (c == "a") then { copy := copy + "a" }
7      else if (c == "b") then { copy := copy + "b" }
8      ...
9      else if (c == "z") then { copy := copy + "z" };
10     i := i + 1
11   };
12   eval(copy)
13 }
```

Fig. 7: Example of an implicit flow through a conditional copy.

$D = (D_{block}, D_{edge})$ as the *dynamic CFG*. Then, if the current block is the initialization of an untrusted input x, we have to store (in V) the initial value of x, that will be potentially checked in future (lines 8–10). If the current block is a string-to-code statement, the contracts verification procedure *Verify* (Algorithm 2 in Fig. 6) is called. It stops the execution if a contract is not satisfied, meaning that the program is potentially under attack (lines 11–13). Finally, the previous block b_p is updated with the current block b, and the language interpreter INTERPRETER executes the instruction associated to the block b and updates the current block and memory (b, μ) (lines $14, 15$).

The *Verify* procedure (Algorithm 2) is the contracts verifier. It is able to stop the execution by throwing an exception if it finds an input not belonging to the language specified in the contract. In particular, the verifier iterates over the set of contracts (line 2) and picks only those contracts ending in the current block b, i.e., $(\mathcal{I}^x, p_0 \dots p_m) \in \mathcal{Contracts}$ such that $p_m = b$ (line 3). Then, it checks whether the path $p_0 \cdots p_m$ is a sub-path of the dynamic CFG D: This is achieved by $m - 1$ calls to REACHABILITY algorithm, returning `true` iff there is a path in D from p_i to p_{i+1} (lines 4–8) for each $0 \leq i < n$. In this case, the procedure checks whether the contract is satisfied by checking if the input value $V(x)$ of the untrusted variable in x, input in the block p_0, is in the language $\mathcal{I}^x$ (lines $9, 10$).

4.4 Handling the implicit information flows

In this section, we show how it is possible to extend the static analysis algorithm in order to track not only the explicit information flows but also the implicit ones. The *explicit information flows* are caused by a direct exchange of information via copy operations, while the *implicit information flows* arise from the control structure of the program [26]. For instance, consider the program in Fig. 7 which performs a *conditional copy* [25] of the input and then evaluates it as code. Its input/output semantic is identical to the program `main(s) { eval(s) }`, but it does not directly copy any bit of the untrusted input into the string-to-code statement (line 12). By only changing the USE function definition, we are able to detect also this kind of attacks: We define the function GUARD(b) which computes the

immediate guard to which the block b is subjected. For instance, in Fig. 7, if we compute GUARD(`copy := copy + "a"`) it returns the set {`c == "a"`}. Hence, we can define the extended function $\widetilde{\text{USE}}(b) = \text{USE}(b) \cup \text{USE}(\text{GUARD}(b))$ which correctly detects also the implicit flow (this technique is based on the notion of *control dependences* [7]).

4.5 Soundness

In Sect. 3, we have provided a model which precisely describes safety against code injection attacks, parametrically on the expected inputs, by embedding the programmer's intention into the definition (i.e., the predicate ϕ_g). However, the semantic aspects of the model and, in particular, the abstract non-interference predicate, make it not suitable to a straightforward implementation. Nonetheless, a semantic definition is fundamental in order to provide an accurate description of the real world problem of code injection.

We now prove the soundness of the proposed approach, w.r.t. an abstract code injection policy $\text{ANI}_P^x(\mathcal{O}, n)$ we aim at enforcing in a program P. The proposed static analysis will capture any vulnerable path by over-approximating them.

Theorem 1 (Soundness of static analysis w.r.t. $\text{ANI}_P^x(\mathcal{O}, n)$**).** *Let* $P \in$ WHILEFUN *be a program,* x *be an untrusted input of* P*,* $n \in \mathcal{L}_P$ *such that* $Stm(n) \in str2code$ *and* $\text{ANI}_P^x(\mathcal{O}, n)$ *an abstract code injection policy, then*

$$\neg\, \text{ANI}_P^x(\mathcal{O}, n) \implies \exists p = b_x \cdots b_n \in \text{REVERSE}(\text{BRANCHES}(Paths(b_n, \varepsilon)))$$

where b_x *and* b_n *denote the blocks containing the untrusted input* x *and the string-to-code statement at program point* n *in the* `main` *procedure, respectively.*

If an untrusted input x interferes with the execution of a string-to-code statement at a program point n, the static analysis will generate a tree rooted in b_n and leading to the leaf b_x, and the programmer will have to specify a contract for the input x concerning the cleaned and reversed path $b_x \cdots b_n$. In that light, the static analysis produces the information used by the monitor to work properly.

We now prove the correctness of the dynamic phase, namely of the *MonInt* algorithm. Its semantics is straightforward: Starting from the initial memory μ_0, it executes the program until a string-to-code statement is not reached. When this happens, if all the contracts on the executed path are satisfied, the statement is executed, otherwise, the execution is stopped throwing an exception.

Theorem 2. *Let* $P \in$ WHILEFUN *be a program. For each initial memory* $\mu_0 \in \mathcal{M}em$*,* $\tau = \llparenthesis\, MonInt \,\rrparenthesis_{(\mathbb{C}, \mathcal{C}ontracts)_P}(\langle 1^1, \mu_0 \rangle)$[5] *implies that*

- τ *is prefix of* $\tau' = \llparenthesis P \rrparenthesis(\langle 1^1, \mu_0 \rangle)$*;*
- $\tau = \llparenthesis P \rrparenthesis(\langle 1^1, \mu_0 \rangle)$ *if and only if for each string-to-code statement executed in* τ *all the contracts are satisfied.*

[5] Execution of the monitor specialized on the annotated program.

In other words, the monitor alters the semantics of the program (by blocking the execution) if and only if at least one input contract, for an untrusted input affecting an executed string-to-code statement, is not satisfied.

Finally, as a corollary to the Theorems 1 and 2, we can set out the following result that justifies our mechanism. Let us define the function associating with each trajectory the sequence of blocks executed in the CFG as $bl(\tau_0 \cdots \tau_n) = bl(\tau_0) \cdots bl(\tau_n)$, where $bl(\langle n^k, \mu \rangle) = b$ if b is the block containing $Stm(n)$.

Corollary 1. *Let $P \in$ WHILEFUN be a program, $n \in \mathcal{L}_P$ such that $Stm(n) \in$ str2code, and $(\mathbb{C}, \mathcal{C}ontracts)_P$ be the static analysis output. Let x be an input of P such that $\exists (\mathcal{I}^x, p_0 \cdots p_m) \in \mathcal{C}ontracts$ with p_0 setting the input x and p_m containing $Stm(n)$*[6]*. Let $v \in \mathcal{V}al^x$, we define $\rho_{\mathfrak{M}}$ as*

$$\rho_{\mathfrak{M}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v] \rangle)) \textit{ iff}$$
$$\forall (\mathcal{I}^x, p) \in \mathcal{C}ontracts.\, p \textit{ sub-trace of } bl(\langle\!| P |\!\rangle^{n^k}(\langle 1^1, \mu_0[x \leftarrow v] \rangle)) \textit{ we have } v \in \mathcal{I}^x$$

Then, given $\phi_{\mathfrak{M}}$ is defined in terms of $\rho_{\mathfrak{M}}$ as in Eq. 1, $\text{MonInt}_{(\mathbb{C}, \mathcal{C}ontracts)_P}(\mu_0)$ enforces the abstract code injection policy: $\forall x$ untrusted input, $\forall v_1, v_2 \in \mathcal{V}al$

$$\phi^x_{\mathfrak{M}}(v_1) = \phi^x_{\mathfrak{M}}(v_2) \implies \rho_{\mathfrak{M}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle) = \rho_{\mathfrak{M}}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle))$$

Note that, our mechanism is not sound "as a matter of principle". It is sound w.r.t. the specification of contracts characterizing safe inputs and output behaviors.

4.6 Complexity considerations

The CFG model and the SSA form are well known code representations and can be computed in polynomial time w.r.t. the abstract syntax tree of the program. The bottleneck of the static phase is the *taint analysis*, computing *Paths* for each string-to-code statement. Unfortunately, the number of these paths could be exponentially large w.r.t. the size of the $\mathbb{C}^{ssa}$. This is due to the generality of our approach, allowing the programmer to provide a contract for each possible vulnerable path, i.e., for each triple (x, p, e) with x untrusted input, p path, and e string-to-code statement. In practice, it is highly unlikely to have an exponential number of ways in which an untrusted input can interfere with a single query execution, therefore we believe that in the average case, this approach scales well. However, the programmer can always reduce the complexity by either providing a (different) contract for each pair (x, e) (reducing complexity to $O(ne)$, n number of the untrusted inputs and e number of string-to-code blocks), or providing a different contract only for each input x (reducing complexity to $O(n)$).

Dynamic monitor worst case checking cost is divided into the cost for computing REACHABILITY between each pair of adjacent nodes (which is polynomial w.r.t. the size of $\mathbb{C}^{ssa}$) for each contract c and checking whether an untrusted input u satisfies the corresponding contract c (whose cost depends on the formalism used to model it).

[6] We consider only one variable for simplicity, but in general we may have more than one untrusted input affecting $Stm(n)$, in this case the generalization is straightforward.

5 Generality of Abstract Code Injection

In this section, we show also by means of two examples, that the definition of abstract code injection that we provide is enough general to cope with the main related works, defining specific notions of code injection [3,24] (a brief summary of these works is given in Sec. 1).

The generality of our approach allows us to detect attacks that elude the mechanisms proposed in the related works: The program given in Fig. 7 is not detected by all the works based on a copy-based taint analysis (as [24]), since that program is built on a pure semantic notion of interference. Furthermore, the flexibility of our mechanism make it more suitable to different types of code injection. For instance, consider the simple program `main(s) { eval(s) }`: all the mechanisms based on the idea of automatically detect a manipulation of the syntactic structure cannot infer nothing about the intended structure, since there are no sufficient information to "guess" the programmer's intention.

Defining code injection attacks (CIAO) [24]. From the definition of code injection attacks given in [24], we can derive the considered set of safe output behaviors $\mathbb{O}_{[24]}$, i.e., all the states reached by a code execution that does not contain any *tainted* (potentially untrusted) *code* symbol[7]. In their paper, they provide an algorithm $A(P, T, U)$[8] to precisely detect what is considered a potential attack, w.r.t. their definition of safe behaviors $\mathbb{O}_{[24]}$. Being (T, U) the tuple of all (trusted and untrusted) inputs, it corresponds to our memory μ_0. We can model their computation of safe behaviors as the characteristic function $\rho_{\mathbb{O}}^{[24]}$ of $\mathbb{O}_{[24]}$, defined as: For each execution point n^k (where a string-to-code statement is executed)

$$\rho_{\mathbb{O}}^{[24]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \overbrace{\mu_0}^{\text{i.e., } (T,U)} \rangle)) \iff A(P, \mu_0) \text{ does not detect an attack at the point } n^k$$

Hence, the algorithm proposed in [24] enforces the abstract code injection policy: $\forall x \in U, v_1, v_2 \in \mathcal{Val}^x$

$$\phi^x_{\mathbb{O}[24]}(v_1) = \phi^x_{\mathbb{O}[24]}(v_2) \implies \rho_{\mathbb{O}}^{[24]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle)) = \rho_{\mathbb{O}}^{[24]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle))$$

where $\phi^x_{\mathbb{O}[24]}$ is defined in terms of $\rho_{\mathbb{O}}^{[24]}$ as in Eq. 1. Hence, from the semantic perspective of our approach, [24] only admits the (abstract) interference that does not cause the execution of tainted code symbols. This is clearly an abstract form of non-interference.

CANDID [3]. In this approach, we can still derive the implicitly used notion of safe output behaviors $\mathbb{O}_{[3]}$ as the set of all the states in which the syntactic structures (i.e., the parsing trees) of each query[9] executed by P on the inputs $i_1, \ldots, i_n$, are equal to the ones produced by the execution of the program P on the *valid representation*[10] of the inputs $i_1, \ldots, i_n$, i.e., $\mathrm{VR}(i_1), \ldots, \mathrm{VR}(i_n)$. Let

[7] In [24], a symbol is considered *code* if it is not a final value.
[8] P is a program, and T and U are the set of trusted and untrusted inputs, respectively.
[9] [3] is focused on the SQLi problem.
[10] A *valid representation* of an input i is a value $\mathrm{VR}(i)$ which is manifestly benign and non-attacking, and it dictates the same path of i in the application.

$B(P, i_1, \ldots, i_n)$ be the function returning the syntactic structure of the query executed at the execution point n^k, we can define the characteristic function $\rho_{\mathbb{O}}^{[3]}$ of $\mathbb{O}_{[3]}$ as

$$\rho_{\mathbb{O}}^{[3]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \overbrace{\mu_0}^{\text{i.e., } (i_1, \ldots, i_n)} \rangle)) \iff B(P, \mu_0) \overbrace{\approx}^{\text{is isomorphic to}} B(P, \mathrm{VR}(\mu_0))$$

Intuitively, in CANDID are safe all the query executions that performed on an input i or on its valid representation $VR(i)$ provide the same execution structure. As before, we can characterize the abstract code injection policy enforced by CANDID as: $\forall x \in U, v_1, v_2 \in \mathcal{V}al^x$

$$\phi_{\mathbb{O}_{[3]}}^x(v_1) = \phi_{\mathbb{O}_{[3]}}^x(v_2) \implies \rho_{\mathbb{O}}^{[3]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_1] \rangle) = \rho_{\mathbb{O}}^{[3]}(\llbracket P \rrbracket^{n^k}(\langle 1^1, \mu_0[x \leftarrow v_2] \rangle)))$$

where, again as before, $\phi_{\mathbb{O}_{[3]}}^x$ is defined in terms of $\rho_{\mathbb{O}}^{[3]}$. Here, the non-admitted interference concerns the alteration of the structure of the parsing tree.

6 Conclusion

In this paper we propose both a general model for abstract code injection policies, i.e., code injection policies parametric on what the programmer considers safe in output, and an algorithmic approach for enforcing abstract code injection policies, based on the combination of a static and a dynamic analysis phase. In particular, the static analysis aids the programmer in finding *what* should be controlled and *when*, i.e., which inputs and which execution paths, have to be checked during execution. The *contracts* that the inputs should meet are asked to the programmer and used to annotate the program. Then a monitor checking the contracts when necessary, namely when a vulnerable path is executed, is proposed. In particular, the application enforcing a given abstract code injection policy consists in the monitor specialized on the annotated program. We finally provide the intuition of the generality of abstract code injection, by showing the abstract injection policies enforced by the main related works.

We tested the feasibility of the monitoring approach by implementing it on a toy language for SQL injection[11], but surely in the future we aim at implementing this analysis approach on real languages. As far as the model is concerned, there are several aspects that deserve further study. In this paper we consider only safe output partitions in safe/unsafe behaviors, but abstract code injection policies could be defined in terms of more precise partitions, providing the possibility of modeling different safety degrees of output behaviors. Finally, in the approach we propose to enforce the policy where the output characterization of safe bahaviors is determined by the input contracts. It would be interesting to find a way for approximating input decidable contracts automatically generated by the output characterization of safe behaviors.

[11] The source code is available for the use at `https://gitlab.com/samuele/KArMA.git`.

References

1. The Open Web Application Security Project (OWASP), https://www.owasp.org/
2. Anley, C.: Advanced sql injection in sql server applications (2002)
3. Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishnan, V.: Candid: preventing sql injection attacks using dynamic candidate evaluations. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 12–24. ACM (2007)
4. Buehrer, G., Weide, B.W., Sivilotti, P.A.: Using parse tree validation to prevent sql injection attacks. In: Proceedings of the 5th international workshop on Software engineering and middleware. pp. 106–113. ACM (2005)
5. Cooper, K., Torczon, L.: Engineering a compiler. Elsevier (2011)
6. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. 277(1-2), 47–103 (2002)
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(4), 451–490 (1991)
8. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. ACM SIGPLAN Notices 39(1), 186–197 (2004)
9. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy, 1982 IEEE Symposium on. pp. 11–11. IEEE (1982)
10. Halfond, W.G., Orso, A.: Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. pp. 174–183. ACM (2005)
11. Halfond, W.G., Orso, A., Manolios, P.: Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 175–185. ACM (2006)
12. Jones, N., Sestoft, P., Søndergaard, H.: An experiment in partial evaluation: The generation of a compiler generator. In: Proc. of the 1st Internat. Conf. on Rewriting techniques and applications. vol. 202, pp. 124–140 (05 1985)
13. Maor, O., Shulman, A.: Sql injection signatures evasion. Imperva, Inc., Apr (2004)
14. Mastroeni, I.: On the rôle of abstract non-interference in language-based security. In: Programming Languages and Systems, Third Asian Symposium, APLAS. pp. 418–433 (2005)
15. Mastroeni, I.: Abstract interpretation-based approaches to security - A survey on abstract non-interference and its challenging applications. arXiv preprint arXiv:1309.5131 (2013)
16. Mastroeni, I., Banerjee, A.: Modelling declassification policies using abstract domain completeness. Mathematical Structures in Computer Science 21(6), 1253–1299 (2011)
17. Mastroeni, I., Zanardini, D.: Abstract program slicing: an abstract interpretation-based approach to program slicing. ACM Transactions on Computational Logic (TOCL) 18(1), 7 (2017)
18. McDonald, S.: Sql injection: Modes of attack, defense, and why it matters. White paper, GovernmentSecurity. org (2002)
19. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. Security and Privacy in the Age of Ubiquitous Computing pp. 295–307 (2005)

20. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
21. OWASP: Top 10 2010. The Ten Most Critical Web Application Security Risks (2010)
22. OWASP: Top 10 2013. The Ten Most Critical Web Application Security Risks (2013)
23. OWASP: Top 10 2017 (release candidate 1). The Ten Most Critical Web Application Security Risks (2017)
24. Ray, D., Ligatti, J.: Defining code-injection attacks. In: ACM SIGPLAN Notices. vol. 47, pp. 179–190. ACM (2012)
25. Ruse, M.E., Basu, S.: Detecting cross-site scripting vulnerability using concolic testing. In: Information Technology: New Generations (ITNG), 2013 Tenth International Conference on. pp. 633–638. IEEE (2013)
26. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications 21(1), 5–19 (2003)
27. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) 3(1), 30–50 (2000)
28. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: ACM SIGPLAN Notices. vol. 41, pp. 372–382. ACM (2006)
29. The Code Curmudgeon: Sql injection hall-of-shame. http://codecurmudgeon.com/wp/sql-injection-hall-of-shame/
30. Wassermann, G., Su, Z.: An analysis framework for security in web applications. In: Proceedings of the FSE Workshop on Specification and Verification of component-Based Systems (SAVCBS 2004). pp. 70–78 (2004)
31. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: USENIX Security Symposium. pp. 121–136 (2006)

A Framework for Computer-Aided Design of Educational Domain Models

Eric Butler, Emina Torlak, and Zoran Popović

University of Washington
{edbutler,emina,zoran}@cs.washington.edu

Abstract. Many educational applications, from tutoring to problem generation, are built on a formal model of the operational knowledge for a given domain. These *domain models* consist of rewrite rules that experts apply to solve problems in the domain; e.g., factoring, $ax+bx \to (a+b)x$, is one such rule for K-12 algebra. Domain models currently take hundreds of hours to create, and they differ widely in how well they meet educational objectives such as maximizing problem-solving efficiency. Rapid, objective-driven creation of domain models is a key challenge in the development of personalized educational tools.

This paper presents RuleSy, a new framework for computer-aided authoring of domain models for educational applications. RuleSy takes as input a set of example problems (e.g., $x+1=2$), a set of basic *axiom* rules for solving these problems (e.g., factoring), and a function expressing the desired educational objective. Given these inputs, it first synthesizes a set of sound *tactic* rules (e.g., combining like terms) that integrate multiple axioms into advanced problem-solving strategies. The axioms and tactics are then searched for a domain model that optimizes the objective. RuleSy is based on new algorithms for mining tactic specifications from examples and axioms, synthesizing tactic rules from these specifications, and selecting an optimal domain model from the axioms and tactics.

We evaluate RuleSy on the domain of K-12 algebra, finding that it recovers textbook tactics and domain models, discovers new tactics and models, and outperforms a prior tool for this domain by orders of magnitude. But RuleSy generalizes beyond K-12 algebra: we also use it to (re)discover proof tactics for propositional logic, demonstrating its potential to aid in designing models for a variety of educational domains.

1 Introduction

A key challenge in the design of educational applications is modeling the operational knowledge that captures the expertise for a given domain. This knowledge takes the form of a *domain model*, which consists of *condition-action* rules that experts apply to solve problems in the domain. For example, factoring, $ax + bx \to (a + b)x$, is one such rule for K-12 algebra: its condition recognizes problem states that trigger rule application, and the action specifies the result. Educational applications rely on domain models to automate tasks such as

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 138–160, 2018.
https://doi.org/10.1007/978-3-319-73721-8_7

problem and progression generation [1], hint and feedback generation [2], student modeling [3], and misconception detection [4].

At present, domain models are created by hand, taking hundreds of hours of development time to model a single hour of instructional material [5]. This limits applications to using one out of many possible models that capture the operational knowledge for a domain. Yet recent research [6] shows that some students need over six times more content than others to master a domain. To best serve a broad population of students, applications therefore need multiple models that optimize different educational objectives [7,8].

To illustrate the difficulty of model authoring, consider creating a domain model for K-12 algebra. Suppose that our model includes the basic rules, or *axioms*, for solving algebra problems: e.g., factoring and constant folding, $c_0 + c_1 \rightarrow c_2$, where c_0 and c_1 are constants and c_2 is their sum. Should this model also include the rule for combining like terms, $c_0x + c_1x \rightarrow c_2x$, which composes factoring and constant folding? While such compound rules, or *tactics*, are redundant with respect to the axioms, standard domain models (e.g., [9]) include them to enable efficient problem solving with fewer steps and less cognitive load [10]. But there is a limit to how many rules students can remember, so the optimal set of axioms and tactics depends on the desired tradeoff between maximizing solving efficiency and minimizing the memorization burden.

This paper presents a new approach for rapid, objective-driven creation of domain models that is based on program synthesis. We realize this approach in RuleSy, a framework that assists developers with creating tactics and domain models that optimize desired objectives. The RuleSy framework was motivated by practical experience: the first and last authors work for Enlearn, an educational technology company building adaptive K12 applications that need custom domain models. Developers of such applications are the intended users of this work, and Enlearn is in the process of adopting key ideas from RuleSy.

RuleSy aids developers by generating an optimal domain model given a set of axioms for the domain, a set of example problems, and an optimization objective expressed in terms of rule and solution costs. Using the axioms and the problems, RuleSy synthesizes an exhaustive set of tactics that combine multiple axioms into advanced problem-solving strategies. Each of these tactics is a sound rule that shortens the solution to at least one example problem compared to using the axioms alone. Following synthesis, RuleSy applies discrete optimization to produce a subset of the axioms and tactics (i.e., a domain model) that both solves the example problems and optimizes the given objective.

RuleSy's algorithms are designed to solve three core technical challenges:

– *Specification.* Synthesizing tactics requires a functional specification of their behavior. Since tactics compose multiple axioms, a sequence of axioms may seem to provide such a specification. For example, we may expect $\mathbf{A} \circ \mathbf{A}$, where $\mathbf{A}$ is the additive identity rule $x + 0 \rightarrow 0$, to describe the tactic $(x+0)+0 \rightarrow x$. But because a condition-action rule can fire on *any* part of the problem state, $\mathbf{A} \circ \mathbf{A}$ describes a set of distinct tactics that also includes, e.g., $x+0 = y+0 \rightarrow x = y$. RuleSy addresses this challenge with a new approach

for extracting tactic functions from the shortest axiom-based solutions to the example problems. Each resulting tactic specification is sound with respect to the axioms, and useful for solving at least one example in fewer steps.

- *Synthesis.* Given a tactic specification, the next challenge is to find a rule that implements it. RuleSy represents rules as programs that operate on problem states expressed as (abstract syntax) trees. Because these trees are unbounded in size, the rule synthesis query cannot be expressed in existing systems for syntax-guided synthesis (e.g., [11,12,13]). To address this challenge, RuleSy employs an efficient new reduction to a set of syntax-guided synthesis queries over (carefully constructed) trees of bounded size. Our reduction exploits the structure of RuleSy's specifications and programs to ensure that the synthesized rules are sound over trees of any size, and to asymptotically reduce the size of the synthesis search space.
- *Optimization.* The final challenge is to search the axioms and tactics for a domain model that both solves the examples and optimizes the input objective. Finding such a model is undecidable in general, since an arbitrary set of condition-action rules (i.e., a candidate model) may not be terminating [14]. RuleSy addresses this challenge with a new algorithm for deciding a more constrained variant of the model optimization task: it finds a domain model that solves the examples while minimizing the objective over the model's rules and the shortest (rather than all) solutions obtainable with those rules.

To evaluate our algorithms, we used RuleSy to model the domain of introductory K-12 algebra, comparing the output to a standard textbook [9] and a prior tool [15] for this domain. Applying RuleSy to examples and axioms from the textbook, we find that it both recovers the tactics presented in the book and discovers new ones. We also find that RuleSy can recover the textbook's domain model, as well as discover variants that optimize different objectives. Finally, we find that RuleSy significantly outperforms the prior tool, both in terms of output quality and runtime performance.

To show that RuleSy generalizes beyond K-12 algebra, we used it to model the domain of propositional logic proofs. Applying RuleSy to textbook [16] axioms and exercises, we find that it synthesizes both new and standard tactics for this domain (e.g., modus ponens), just as it did for K-12 algebra.

The rest of the paper is organized as follows. Section 2 illustrates RuleSy on a toy algebra domain. Section 3 describes RuleSy's language of condition-action rules. Section 4 describes the new algorithms for specification mining, rule synthesis, and model optimization. Section 5 presents our two case studies. Section 6 reviews related work, and Section 7 concludes.

2 Overview

This section illustrates RuleSy's functionality on a toy algebra domain. We show the specifications, tactics, and models that RuleSy creates for this domain, given a set of example problems, axioms, and an objective.

2.1 Examples, Axioms, and Objectives

Figure 1 shows our example problems, axioms, and objective for toy algebra.

Examples. The problems (1b) are represented as syntax trees. We consider a tiny subset of algebra that includes equations of the form $x + \sum_i c_i = c_k$, where x is a variable and c_i, c_k are integer constants. Experts solve these problems by applying condition-action rules until they obtain an equation of the form $x = c$.

Axioms. The axioms (1a) are expressed as programs in the RuleSy language (Section 3). A rule program consists of a *condition*, which matches a syntax tree with a specific shape, and an *action*, which creates a new tree by applying editing operations (such as adding or removing nodes) to the matched tree. For example, the axiom **A** matches trees of the form $(+\ 0\ e \ldots)$, where the order of subtrees is ignored, and it rewrites such trees by removing the constant 0 to produce $(+\ e \ldots)$. The shown axioms can solve all problems in the toy algebra domain. For example, we can solve p_1 in two steps by applying $\mathbf{B} \circ \mathbf{A}$ to obtain $x + 1 + -1 = 5 \rightarrow_{\mathbf{B}} x + 0 = 5 \rightarrow_{\mathbf{A}} x = 5$. RuleSy uses the axioms to synthesize tactic rules (Figure 3) that can solve the example problems in fewer steps.

Objective. The educational objective (1c) is expressed as a function of rule and solution costs. Rule cost measures the complexity of a rule's syntactic representation. Solution cost measures the efficiency of a solution in terms of the tree edits performed to solve an example problem. These costs are proxy measures for the difficulty of learning and applying knowledge in a given domain model [10]. RuleSy selects a domain model that best balances the trade-off between rule complexity and solution efficiency specified by the objective.

2.2 Specifications, Tactics, and Domain Models

Specifications. To help with domain modeling, RuleSy first needs to synthesize a set of useful tactics, which can solve the input problems more efficiently than the axioms alone. For example, we could solve p_1 in one step if we had a "cancelling opposite constants" tactic that composes the axioms **B** and **A**. RuleSy determines which rules to synthesize, and how those rules should behave, by mining tactic specifications (Section 4.1) from the shortest solutions to the example problems that are obtainable with the axioms (Figure 2a).

A RuleSy specification takes the form of a *plan* for applying a sequence of axioms (Figure 2b). A plan describes which axioms to apply, in what order, and how. Since an axiom may be applied to a problem in multiple ways, a plan associates each axiom with an application index and a binding for the axiom's pattern. The application index identifies a subtree in the expression's abstract syntax tree (AST), and the binding specifies a mapping from the index space of the axiom's pattern to the index space of the subtree. The plan in Figure 2b specifies a generic tactic for cancelling opposite constants; for example, it solves p_1 in one step by reducing the expression $(+\, x\, 1\, {-1})$ to x (Figure 2c). In essence, plans

```
(define A ; Additive identity: (+ 0 e ...) → (+ e ...)
  (Rule (Condition (Pattern (Term + (ConstTerm) _ etc))
                   (Constraint (Eq? (Ref 1) 0)))
        (Action    (Remove (Ref 1)))))

(define B ; Constant folding: (+ c1 c2 ...) → (+ c ...), c = c1 + c2
  (Rule (Condition (Pattern (Term + (ConstTerm) (ConstTerm) etc))
                   (Constraint true))
        (Action    (Replace (Ref 1) (Apply + (Ref 1) (Ref 2)))
                   (Remove (Ref 2)))))

(define C ; Adding opposite: (= (+ e0 ...) e1) → (= (+ (- e0) e0 ...) (+ e1 (- e0)))
  (Rule (Condition (Pattern (Term = (Term + _ etc) _))
                   (Constraint true))
        (Action    (Replace (Ref 1) (Cons (Make - (Ref 1 1)) (Ref 1)))
                   (Replace (Ref 2) (Make + (Ref 2) (Make - (Ref 1 1)))))))
```

(a) Axioms in the RuleSy language (Section 3).

```
; Problem p0 : x + 0 = 3
(= (+ x 0) 3)

; Problem p1 : x + 1 + -1 = 5
(= (+ x 1 -1) 5)

; Problem p2 : x + 2 = -4
(= (+ x 2) -4)
```

(b) Example problems.

$$f(\mathcal{R}, \mathcal{G}) = \alpha \sum_{R \in \mathcal{R}} \mathit{RuleCost}(R) + (1-\alpha) \frac{\sum_{G \in \mathcal{G}} \mathit{SolCost}(G)}{|\mathcal{G}|}$$

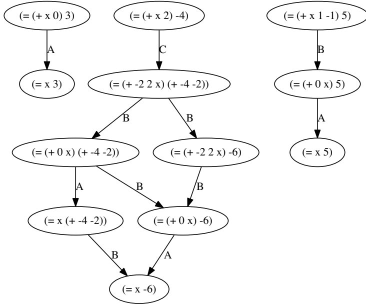
(c) A sample objective function, where $\alpha \in [0, 1]$, and $\mathcal{G}$ contains the shortest solutions obtained with the rules $\mathcal{R}$.

Fig. 1: The inputs to RuleSy for the toy algebra domain.

are functional specifications of tactics that can help solve the example problems in fewer steps—and that are amenable to sound and efficient synthesis.

Tactics. Given a set of plans, RuleSy synthesizes the corresponding tactics (Section 4.2), expressed in the same language (Section 3) as the input axioms. Figure 3 shows two sample tactics synthesized for the plans (e.g., Figure 2b) mined from the toy examples and axioms. These tactics perform fewer tree edits than the axiom sequences they replace, leading to cheaper solutions. For example, the tactic **BA** performs two tree edits, while the axiom sequence $\mathbf{B} \circ \mathbf{A}$ performs three such edits. But the tactic also applies to fewer problem states than the axioms. RuleSy uses discrete optimization, guided by the input objective, to decide which axioms and tactics to include in a domain model.

Domain Models. The RuleSy optimizer (Section 4.3) searches the axioms and tactics for a domain model that is sufficient to solve the example problems, while minimizing the objective over all shortest solutions obtainable with such models. Figure 4 shows two sample optimal models for the toy algebra domain. The models $\mathcal{R}_{0.1}$ and $\mathcal{R}_{0.9}$ minimize the toy objective (Figure 1c) for different values of the weighting factor α (0.1 and 0.9, respectively). The model $\mathcal{R}_{0.1}$ includes more rules because lower values of α emphasize solution efficiency over domain model economy. RuleSy helps with rapid navigation of such design tradeoffs.



(a) All shortest solutions for the toy algebra problems and axioms (Figure 1).

$$[\langle \mathbf{B}, [], \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [3]\}\rangle,$$
$$\langle \mathbf{A}, [], \{[] \mapsto [], [1] \mapsto [1]\}\rangle]$$

(b) A plan for applying the axioms $\mathbf{B} \circ \mathbf{A}$.

Input	(= (+ x 1 -1) 5)	(= (+ 0 x) 5)
Axiom	**B**	**A**
Binding	(= (+ 1 -1 x) 5)	(= (+ 0 x) 5)
Output	(= (+ 0 x) 5)	(= x 5)

(c) Using the plan in (b) to solve the problem p_1 (Figure 1b).

Fig. 2: A sample plan (b) mined from the shortest solutions (a) to the toy algebra problems. The plan specifies a tactic for canceling opposite constants (c).

```
; Canceling opposite constants: (+c −c e...) → (+ e ...).
(define BA
  (Rule (Condition (Pattern (Term + (ConstTerm) (ConstTerm) _ etc))
                   (Constraint (Eq? (Ref 1) (Apply - (Ref 2)))))
        (Action    (Remove (Ref 1))
                   (Remove (Ref 2)))))

; Move negated constant to other side with only one other term:
; (= (+ c1 e ...) c2) → (= (+ e ...) c), c = c2 − c1.
(define CBAB
  (Rule (Condition (Pattern (Term = (Term + (ConstTerm) _ etc) (ConstTerm)))
                   (Constraint true))
        (Action    (Remove (Ref 1 1) 0)
                   (Replace (Ref 2) (Apply - (Ref 2) (Ref 1 1))))))
```

Fig. 3: Sample tactics synthesized for the toy plans (e.g., Figure 2b).

3 A Language for Condition-Action Rules

This section presents the RuleSy language for specifying condition-action rules. The language is parametric in its definition of problem states. For concreteness, we present an instantiation of RuleSy for the domain of K-12 algebra. We describe another instantiation, for the domain of propositional logic, in Section 5.

Specifying Problems and Rules. The RuleSy domain-specific language (DSL) for algebra represents rules as *programs* that operate on problems expressed as *terms* (Figure 5a). RuleSy is parametric in the definition of terms, but the structure of rules is fixed. A rule consists of a *condition*, which determines if the rule is applicable to a given term, and an *action*, which specifies how to transform

$$\mathcal{R}_{0.1} = \{\mathbf{A}, \mathbf{BA}, \mathbf{CBAB}\}\ \mathcal{R}_{0.9} = \{\mathbf{BA}, \mathbf{CBAB}\}$$

Fig. 4: Optimal domain models for toy algebra (Figures 1 and 3).

```
program  := (Rule cond action)
   cond  := (Condition (Pattern pattern)
         :=            (Constraint constr))
pattern  := _ | (ConstTerm) | (VarTerm) | (BaseTerm)
          | (Term op pattern+)
          | (Term op pattern+ etc)
 constr  := true | pred | (And constr constr)
   pred  := (Eq? ref const) | (Neq? ref const)
 action  := (Action cmd+)
    cmd  := (Remove ref) | (Replace ref expr)

   expr  := const | obj
    obj  := (Make op expr+)
          | (Cons expr ref)
          | (Cons expr obj)
  const  := int | ref | (Apply op const+)
    ref  := (Ref) | (Ref int+)
   term  := int | var | (op term+)
    int  := integer literal
    var  := identifier
     op  := + | - | * | / | =
```

(a) Syntax for the algebra DSL.

$$
\begin{aligned}
[\![\texttt{(Rule}\, c\, a\texttt{)}]\!]t &= \textbf{if } [\![c]\!]t \textbf{ then } [\![a]\!]t \textbf{ else } \bot \\
[\![\texttt{(Condition}\, p\, b\texttt{)}]\!]t &= [\![p]\!]t \wedge [\![b]\!]t \\
[\![\texttt{(Pattern}\, p\texttt{)}]\!]t &= [\![p]\!]t \\
[\![\texttt{(Constraint}\, b\texttt{)}]\!]t &= [\![b]\!]t \\
[\![\texttt{(Term}\, o\, p_1 \ldots p_k\texttt{)}]\!]t &= (t = (o\ t_1 \ldots t_k)) \wedge \\
&\quad\ \forall_{1 \le i \le k} [\![p_i]\!]t_i \\
[\![\texttt{(Term}\, o\, p_1 \ldots p_k\ \texttt{etc)}]\!]t &= (t = (o\ t_1 \ldots t_n)) \wedge \\
&\quad\ n \ge k \wedge \forall_{1 \le i \le k} [\![p_i]\!]t_i \\
[\![\texttt{(ConstTerm)}]\!]t &= \mathit{literal}(t) \\
[\![\texttt{(VarTerm)}]\!]t &= \mathit{variable}(t) \\
[\![\texttt{(BaseTerm)}]\!]t &= \mathit{literal}(t) \vee \mathit{variable}(t)
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{_}]\!]t &= \text{true} \\
[\![\texttt{true}]\!]t &= \text{true} \\
[\![\texttt{(Eq?}\, r\, e\texttt{)}]\!]t &= ([\![r]\!]t = [\![e]\!]t) \\
[\![\texttt{(Neq?}\, r\, e\texttt{)}]\!]t &= ([\![r]\!]t \ne [\![e]\!]t) \\
[\![\texttt{(And}\, b_1\, b_2\texttt{)}]\!]t &= [\![b_1]\!]t \wedge [\![b_2]\!]t \\
[\![\texttt{(Action}\, a_1 \ldots a_k\texttt{)}]\!]t &= ([\![a_1]\!] \,\|\, \ldots \,\|\, [\![a_k]\!])(t) \\
[\![\texttt{(Remove}\, r\texttt{)}]\!]t &= \mathit{rm}(t, \mathit{index}(r)) \\
[\![\texttt{(Replace}\, r\, e\texttt{)}]\!]t &= \mathit{replace}(t, \mathit{index}(r), [\![e]\!]t) \\
[\![\texttt{(Make}\, o\, e_1 \ldots e_k\texttt{)}]\!]t &= (o\ [\![e_1]\!]t\ \ldots\ [\![e_k]\!]t) \\
[\![\texttt{(Cons}\, e_1\, e_2\texttt{)}]\!]t &= \mathit{cons}([\![e_1]\!]t, [\![e_2]\!]t) \\
[\![\texttt{(Apply}\, o\, e_1 \ldots e_k\texttt{)}]\!]t &= [\![o]\!]([\![e_1]\!]t, \ldots, [\![e_k]\!]t) \\
[\![\texttt{(Ref}\, i_1 \ldots i_k\texttt{)}]\!]t &= \mathit{ref}(t, [i_1, \ldots, i_k])
\end{aligned}
$$

$$
\begin{aligned}
\mathit{index}(\texttt{(Ref}\, i_1 \ldots i_k\texttt{)}) &= [i_1, \ldots, i_k] \\
\mathit{replace}(t, [], s) &= s \\
\mathit{replace}((o\ t_1 \ldots t_k), [i], s) &= (o\ t_1\ \ldots\ t_{i-1}\ s\ t_{i+1} \ldots\ t_k) \\
\mathit{replace}((o\ t_1 \ldots t_k), [i, j, \ldots], s) &= (o\ t_1 \ldots \mathit{replace}(t_i, s, [j, \ldots]) \ldots t_k) \\
\mathit{rm}((o\ t_1 \ldots t_k), [i]) &= (o\ t_1\ \ldots\ t_{i-1}\ t_{i+1} \ldots\ t_k) \\
\mathit{rm}((o\ t_1 \ldots t_k), [i, j, \ldots]) &= (o\ t_1\ \ldots\ \mathit{rm}(t_i, [j, \ldots])\ \ldots\ t_k) \\
\mathit{cons}(t, (o\ t_1 \ldots t_k)) &= (o\ t\ t_1\ \ldots\ t_k)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fire}(R, t) &= \{[\![R]\!]t \mid [\![R]\!]t \ne \bot\} \textbf{ where } \mathit{literal}(t) \vee \mathit{variable}(t) \\
\mathit{fire}(R, t) &= \{[\![R]\!]t^{\beta} \mid [\![R]\!]t^{\beta} \ne \bot \wedge \beta \in \mathcal{B}(\mathit{pattern}(R), t)\} \cup \bigcup_{1 \le i \le n} \{\mathit{replace}(t, [i], s) \mid s \in \mathit{fire}(R, t_i)\} \\
&\quad \textbf{where } t = (o\ t_1 \ldots t_n)
\end{aligned}
$$

(b) Semantics for the algebra DSL. The expression $(o\ t_1\ \ldots\ t_n)$ constructs a term with the given operator and children; $\|$ stands for parallel function composition; $[x, \ldots]$ is a sequence; and other notation is described in Definitions 1-4.

Fig. 5: Syntax (a) and semantics (b) for the RuleSy algebra DSL.

terms. Conditions include a *pattern* to match against the term's structure and a boolean *constraint* to evaluate on that structure. Actions are sequences of term editing operations, such as removing or replacing a subterm. Both constraints and actions can use *references* to identify specific subterms of the term to which the rule is being applied. The toy problems (Figure 1b) and rules (Figures 1a and 3) from Section 2 are all valid terms and programs in the algebra DSL.

Semantics of Rule Firing. Rule programs denote partial functions from terms to terms (Figure 5b). If a term satisfies the rule's condition, the result is a term; otherwise, the result is an undefined value ($\bot$). We solve problems by applying rules exhaustively, via $\mathit{fire}(R, t)$, on permutations and subterms of a given term. Permuting a term (Definition 2) reorders the arguments to any commutative operators while leaving the rest of the term's structure unchanged. To fire a rule on a term, we permute the term "just enough" to establish a one-to-one mapping

between the rule's pattern and the term's structure (Definition 4). By establishing such mappings for all subterms of a term, *fire* implements the intuitive notion of rule application given in Section 2: a rule fires on all subterms that satisfy the rule's condition, ignoring the order of arguments to commutative operators.

To illustrate the semantics of *fire*, consider firing the rule **A** from Figure 1a on the term $t = (+\ (*\ x\ 2)\ 0)$. The set $refs(t)$ of all valid tree indices for t consists of the indices $[], [1], [1,1], [1,2], [2]$, which identify the subterms $t, (*\ x\ 2), x, 2, 0$, respectively (Definition 1). Since both $+$ and $*$ are commutative, valid tree permutations $\Pi(t)$ for t consist of the following mappings (Definition 2):

$$t^{\pi_0} = t \qquad \pi_0 = \{[] \mapsto [], [1] \mapsto [1], [1,1] \mapsto [1,1], [1,2] \mapsto [1,2], [2] \mapsto [2]\}$$
$$t^{\pi_1} = (+\ (*\ 2\ x)\ 0) \qquad \pi_1 = \{[] \mapsto [], [1] \mapsto [1], [1,1] \mapsto [1,2], [1,2] \mapsto [1,1], [2] \mapsto [2]\}$$
$$t^{\pi_2} = (+\ 0\ (*\ x\ 2)) \qquad \pi_2 = \{[] \mapsto [], [1] \mapsto [2], [1,1] \mapsto [2,1], [1,2] \mapsto [2,2], [2] \mapsto [1]\}$$
$$t^{\pi_3} = (+\ 0\ (*\ 2\ x)) \qquad \pi_3 = \{[] \mapsto [], [1] \mapsto [2], [1,1] \mapsto [2,2], [1,2] \mapsto [2,1], [2] \mapsto [1]\}$$

Next, we observe that the scope (Definition 3) of **A**'s pattern consists of the indices $\{[], [1], [2]\}$. Finally, we use the permutations of t and the scope of **A** to compute all valid bindings for **A** and t (Definition 4): $\beta_0 = \{[] \mapsto [], [1] \mapsto [1], [2] \mapsto [2]\}$ with $t^{\beta_0} = t^{\pi_0}$, and $\beta_1 = \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [1]\}$ with $t^{\beta_1} = t^{\pi_2}$. The rule **A** applies only to t^{β_1}, so $fire(\mathbf{A}, t)$ yields $\{(*\ x\ 2)\}$.

Definition 1 (Tree Indices). *A* tree index *is a finite sequence of positive integers that identifies a subterm of a term as follows:* $ref(t, []) = t$; $ref(\texttt{(}o\ t_1 \ldots t_k\texttt{)}, [i]) = t_i$ *if* $1 \le i \le k$; $ref(\texttt{(}o\ t_1 \ldots t_k\texttt{)}, [i, j, \ldots]) = ref(t_i, [j, \ldots])$ *if* $1 \le i \le k$; $ref(t, idx) = \bot$ *otherwise. We write* $refs(t)$ *for the set* $\{idx \mid ref(t, idx) \neq \bot\}$.

Definition 2 (Tree Permutations). *A function* π *is a* tree permutation *for a term* t *if it defines a bijective mapping from* $refs(t)$ *to itself. A permutation* π *is* valid *for* t *if it reorders only the children of commutative operators in* t. *That is, for each* $[i_1, \ldots, i_n] \in refs(t)$, $\pi([i_1, \ldots, i_n]) = [j_1, \ldots, j_n]$ *such that* $\pi([i_1, \ldots, i_{n-1}]) = [j_1, \ldots, j_{n-1}]$ *and* $i_n = j_n$ *or* $ref(t, [i_1, \ldots, i_{n-1}]) = (op\ \ldots)$ *where op is commutative. We write* $\Pi(t)$ *to denote the set of all valid permutations of* t, *and* t^π *to denote the term obtained by applying* $\pi \in \Pi(t)$ *to* $refs(t)$.

Definition 3 (Scopes). *A tree index idx is in the* scope *of a pattern* p *if* $scope(p, idx) \neq \bot$ *where:* $scope(p, []) = p$; $scope(\texttt{(Term}\ o\ p_1 \ldots p_k\texttt{)}, [i]) = p_i$ *if* $1 \le i \le k$; $scope(\texttt{(Term}\ o\ p_1 \ldots p_k\texttt{)}, [i, j, \ldots]) = scope(p_i, [j, \ldots])$ *if* $1 \le i \le k$; $scope(\texttt{(Term}\ o\ p_1 \ldots p_k\ \texttt{etc)}, idx) = scope(\texttt{(Term}\ o\ p_1 \ldots p_k\texttt{)}, idx)$; *and* $scope(p, idx) = \bot$ *otherwise. We write* $scope(p)$ *to denote the set* $\{idx \mid scope(p, idx) \neq \bot\}$.

Definition 4 (Bindings). *Let* β *be a bijection from tree indices to tree indices with a finite domain* $dom(\beta)$ *and range* $ran(\beta)$. *We say that* β *is a* binding *for a pattern* p *if the domain of* β *is the scope of* p; *i.e.,* $dom(\beta) = scope(p)$. *A binding* β *is* valid *for a term* t *if there is a permutation* $\pi \in \Pi(t)$ *such that* $\beta^{-1} \subseteq \pi$ *and for all* $[i_1, \ldots, i_n] \in refs(t)$, *if* $[i_1, \ldots, i_k] \in ran(\beta)$ *and* $[i_1, \ldots, i_{k+1}] \notin ran(\beta)$, *then* $\pi([i_1, \ldots, i_n]) = \beta^{-1}[i_1, \ldots, i_k] \oplus [i_{k+1}, \ldots, i_n]$, *where* $\oplus$ *stands for sequence concatenation. We define* $bind(\beta, t)$ *to return an arbitrary but deterministically chosen permutation* $\pi \in \Pi(t)$ *for which* β *is valid, if one exists, or* $\bot$ *otherwise. We write* $\mathcal{B}(p, t)$ *to denote the set* $\{\beta \mid dom(\beta) = scope(p) \wedge bind(\beta, t) \neq \bot\}$ *of all valid bindings for* p *and* t, *and we write* t^β *to denote* $t^{bind(\beta, t)}$.

Semantics of Conditions and Actions. Rule conditions denote functions from terms to booleans, and actions are functions from terms to terms. A condition maps a term to 'true' if the term matches the condition's pattern and satisfies its constraint. Constraints capture conditions that are not expressible through pattern matching, such as two subterms being syntactically equal. Actions apply a set of parallel functional edits to disjoint subterms of the input term t. Actions can create new terms (via `Make`), and both conditions and actions can evaluate expression terms with literal arguments (via `Apply`).

Well-formed Rule Programs. The meaning of rule conditions and actions is defined only for *well-formed programs* (Definition 5), which contain no invalid references. A reference expression (`Ref` $i_1 \dots i_n$) specifies an index $[i_1, \dots, i_n]$ into the matched term's abstract syntax tree (Definition 1). If a term matches the pattern of a well-formed program, then every reference in that program specifies a valid index into the term's AST. Additionally, `Apply` and `Cons` expressions reference subterms of the right kind; the program's actions edit disjoint subtrees of the term's AST; and (in)equality predicates only compare subterms matched by terminal patterns. RuleSy consumes and creates only well-formed programs.

Definition 5 (Well-Formed Programs). *Let R be a rule with the condition* (`Condition` (`Pattern` p) (`Constraint` b)) *and action* (`Action` $a_1 \dots a_n$)*. We say that R is* well-formed *if the following constraints hold:*

- $index(r) \in scope(p)$ *for all references r in R.*
- $scope(p, index(r)) =$ (`ConstTerm`) *for all references r in all* `Apply` *expressions.*
- $scope(p, index(r)) =$ (`Term ...`) *for all* (`Cons` e r) *expressions.*
- $scope(p, index(r)) \neq$ (`Term ...`) *for all references r in all* `Eq?`, `Neq?` *predicates.*
- *Let r_k denote the first argument to a command a_k in R. For all distinct a_i, a_j in R, $index(r_i)$ is not a prefix of $index(r_j)$ and vice versa.*

4 Rule Mining, Synthesis, and Optimization

Given an educational objective, example problems, and axioms for solving those problems, RuleSy produces an optimal domain model in three stages: (1) specification mining, (2) rule synthesis, and (3) domain model optimization. This section presents the algorithms underlying each stage and states their guarantees. Proofs of these statements are available in our technical report on RuleSy [17].

4.1 Specification Mining

Specification mining takes as input a set of examples and axioms, and produces a set of specifications for tactic rules. We describe the key challenge in specifying tactics; show how our notion of *execution plans* addresses it; and present our FindSpecs algorithm for computing these plans.

Specifying Tactics. To enable efficient synthesis of useful rules, a tactic specification should capture the semantics of a rule—i.e., a partial function—that

can help solve some problems in fewer steps than the axioms alone. But natural forms of specification, such as axiom sequences, do not satisfy this requirement. To see why, consider the axiom sequence $\mathbf{I} \circ \mathbf{B}$, where $\mathbf{I}$ implements factoring (Figure 6) and $\mathbf{B}$ implements constant folding (Figure 1a). Intuitively, we would like $\mathbf{I} \circ \mathbf{B}$ to specify the tactic **IB** for combining like terms (Figure 6). Yet no interpretation of this sequence captures the meaning of the tactic. If we interpret $\mathbf{I} \circ \mathbf{B}$ using the *fire* semantics, the result is a non-functional relation that includes the meaning of multiple tactics. For example, firing $\mathbf{I} \circ \mathbf{B}$ on the term $(+\,(*\,2\,x)\,(*\,3\,x)\,(*\,4\,y)\,(*\,5\,y))$ produces both $(+\,(*\,5\,x)\,(*\,4\,y)\,(*\,5\,y))$ and $(+\,(*\,9\,y)\,(*\,2\,x)\,(*\,3\,x))$. But if we interpret $\mathbf{I} \circ \mathbf{B}$ as the composition of the partial functions denoted by its axioms—i.e., as $\lambda t.[\![\mathbf{B}]\!]([\![\mathbf{I}]\!]t)$—the resulting relation is empty and thus fails to specify a useful tactic.

```
(define I ; (+ (* e0 e) (* e1 e) ...) → (+ (* (+ e0 e1) e) ...)
  (Rule (Condition (Pattern (Term + (* _ _) (* _ _) etc))
                   (Constraint (Eq? (Ref 1 2) (Ref 2 2))))
        (Action    (Remove (Ref 1))
                   (Replace (Ref 2 1) (Make + (Ref 1 1) (Ref 2 1))))))

(define IB ; (+ (* c0 e) (* c1 e) ...) → (+ (* c e) ...), c = c0 + c1
  (Rule (Condition (Pattern (Term + (* (ConstTerm) _) (* (ConstTerm) _) etc))
                   (Constraint (Eq? (Ref 1 2) (Ref 2 2))))
        (Action    (Remove (Ref 1))
                   (Replace (Ref 2 1) (Apply + (Ref 1 1) (Ref 2 1))))))
```

Fig. 6: The tactic **IB** for combining like terms combines factoring (**I**) and constant folding (**B** in Figure 1a), but no interpretation of $\mathbf{I} \circ \mathbf{B}$ captures its behavior.

Execution Plans. We address the challenge of specifying tactic rules with *execution plans.* An execution plan (Definition 7) is a partial function from terms to terms, encoded as a sequence of execution steps (Definition 6). An execution step combines a rule R with a tree index idx and a binding β for R's pattern. The step $\langle R, idx, \beta\rangle$ uses the binding β, if it is valid for the subterm $ref(t, idx)$ of a term t, to evaluate the rule R. An execution step thus specifies where to apply a rule (i.e., to which subterm of a term) and how (i.e., to which permutation of the subterm), while an execution plan composes a sequence of such rule applications. For example, the plan $[\langle \mathbf{I}, [], \beta_0\rangle, \langle \mathbf{B}, [1,1], \beta_0\rangle]$, where β_0 denotes the identity binding, captures the behavior of the combine-like-terms rule on terms of the form $(+\,(*\,c_0\,e)\,(*\,c_1\,e)\,\ldots)$. Moreover, firing a program that implements this plan (e.g., **IB**) captures the common understanding of what it means to combine like terms when solving algebra problems. Execution plans thus satisfy our requirement for tactic specifications by defining useful functional relations.

Definition 6 (Execution Step). *An execution step $\langle R, idx, \beta\rangle$ consists of a rule program R, tree index idx, and a binding β for R's pattern. A step denotes a partial function over terms where $[\![\langle R, idx, \beta\rangle]\!]t = replace(t, idx, [\![R]\!]s^{\beta})$ if $s = ref(t, idx)$, $\beta \in \mathcal{B}(pattern(R), s)$, and $[\![R]\!]s^{\beta} \neq \bot$; otherwise, $[\![\langle R, idx, \beta\rangle]\!]t = \bot$.*

Definition 7 (Execution Plan). *An execution plan S is a finite sequence of execution steps $[\langle R_1, idx_1, \beta_1\rangle, \ldots, \langle R_n, idx_n, \beta_n\rangle]$. The plan $\mathcal{S}$ composes its steps as follows: $[\![S]\!]t_0 = t_n$ if $[\![\langle R_i, idx_i, \beta_i\rangle]\!]t_{i-1} = t_i$ and $t_i \neq \bot$ for all $1 \leq i \leq n$; otherwise, $[\![S]\!]t_0 = \bot$. The plan S is* general *if the step indices $idx_1, \ldots, idx_n$ have the empty index [] as their greatest common prefix.*

Computing Plans. RuleSy mines execution plans from a set of example problems and axioms using the FindSpecs procedure shown in Figure 7. FindSpecs first obtains a *solution graph* (Definition 8) of all shortest solutions to each example problem (line 3). It then applies the FindPlan procedure to compute an execution plan for every path between every pair of nodes in each resulting graph (line 6). These plans specify the set of sound partial functions (Definition 10) that can shorten the solution to at least one example problem (Theorem 1).

```
1: function FindSpecs(T: set of terms, A: set of well-formed programs)
2:   S ← {}
3:   for all ⟨N, E⟩ ∈ {Solve(t, A) | t ∈ T} do
4:     for all src, tgt ∈ N do
5:       paths ← allPaths(src, tgt, ⟨N, E⟩)          ▷ All paths from src to tgt
6:       S ← S ∪ {⟨FindPlan(p), src, tgt⟩ | p ∈ paths ∧ |p| > 1}
7:   return S                                       ▷ Execution plans for T and A

8: function FindPlan(p : n_0 →_{R_1} n_1 →_{R_2} ... →_{R_k} n_k)
9:   S ← an empty array of size k with indices starting at 1
10:  for all 1 ≤ i ≤ k do
11:    idx, β ← firingParameters(R_i, n_{i−1}, n_i)
12:    S[i] ← ⟨R_i, idx, β⟩
13:  root ← greatestCommonPrefix({idx | ⟨R, idx, β⟩ ∈ S})
14:  for all 1 ≤ i ≤ k do               ▷ Drop the common prefix from all indices
15:    ⟨R, idx, β⟩ ← S[i]
16:    S[i] ← ⟨R, dropPrefix(idx, root), β⟩
17:  return S                           ▷ A general execution plan for replaying p
```

Fig. 7: FindSpecs takes as input a set of example problems T and axioms $\mathcal{A}$, and produces a set of plans $\mathcal{S}$ for composing the axioms into tactics.

Definition 8 (Solution Graph). *A directed multigraph $G = \langle N, E\rangle$ is a* solution graph *for a term t, predicate* Reduced*, and rules $\mathcal{R}$ if $t \in N$; E is a set of labeled edges $\langle src, tgt\rangle_R$ such that $src, tgt \in N$, $R \in \mathcal{R}$, and $tgt \in fire(R, src)$; G is acyclic; t is the only term in G with no incoming edges; G contains at least one sink term with no outgoing edges; and each sink satisfies the* Reduced *predicate.*

FindPlan takes as input a path p in a solution graph and produces a general execution plan (Definition 7) for *replaying* that path (Definition 9). The first loop, at lines 10-12, creates a plan that replays the path p from n_0 to n_k exactly: i.e., $[\![S]\!]n_0 = n_k$. The function *firingParameters* (line 11) returns the parameters used to *fire* the rule R_i on n_{i-1} to produce n_i. These include the index *idx* of the subterm to which R_i was applied, as well as the binding β for permuting that subterm. The resulting execution step (line 12) thus reproduces the edge $\langle n_{i-1}, n_i\rangle_{R_i}$: $[\![\langle R_i, idx, \beta\rangle]\!]n_{i-1} = n_i$. The second loop, at lines 13–16, generalizes S to be more widely applicable, while still replaying the path p.

Definition 9 (Replaying Paths). *Let $p = n_0 \rightarrow_{R_1} \ldots \rightarrow_{R_k} n_k$ be a path in a solution graph, consisting of a sequence of k edges labeled with rules $R_1, \ldots, R_k$. An execution plan S* replays *the path p if S is a sequence of k steps $[\langle R_1, idx_1, \beta_1\rangle, \ldots, \langle R_k, idx_k, \beta_k\rangle]$, one for each edge in p, and there is an index $idx \in refs(n_0)$ such that $n_k = replace(n_0, idx, [\![S]\!]ref(n_0, idx))$.*

To illustrate, consider applying FindPlan to the path (= (+ x 1 -1) 5) $\rightarrow_{\mathbf{B}}$ (= (+ 0 x) 5) $\rightarrow_{\mathbf{A}}$ (= x 5) in Figure 2a. The Solve procedure computes this path p by firing **B** with $idx = [1]$, $\beta_{\mathbf{B}} = \{[] \mapsto [], [1] \mapsto [2], [2] \mapsto [3]\}$, and **A** with $idx = [1]$, $\beta_{\mathbf{A}} = \{[] \mapsto [], [1] \mapsto [1]]\}$. As a result, the loop at lines 10-12 executes twice to produce the plan $S = [\langle \mathbf{B}, idx, \beta_{\mathbf{B}}\rangle, \langle \mathbf{A}, idx, \beta_{\mathbf{A}}\rangle]$. The plan S replays p exactly: it describes a tactic for applying the axioms $\mathbf{B} \circ \mathbf{A}$ to a term whose first child has two opposite constants as its second and third children. The loop at lines 13-16 generalizes S to produce the plan in Figure 2b. This plan replays p but applies to *any* term with opposite constants as its second and third children.

Definition 10 (Soundness). *Let f be a partial function from terms to terms. We say that f is* sound *with respect to a set of rules $\mathcal{R}$ if for every term t_0, $f(t_0) = \bot$ or there is a finite sequence of terms $t_1, \ldots, t_k$ such that $f(t_0) = t_k$ and $\forall i \in \{1, \ldots, k\}. \exists R \in \mathcal{R}. t_i \in fire(R, t_{i-1})$.*

Definition 11 (Shortcuts). *A path p is a* shortcut path *in a solution graph G if p contains more than one edge and p is a subpath of a shortest path from G's source to one of its sinks.*

Theorem 1. *Let T be a set of terms,* Reduced *a predicate over terms, and $\mathcal{A}$ a set of rules. If every term in T can be* Reduced *using $\mathcal{A}$, then* FindSpecs$(T, \mathcal{A})$ *terminates and produces a set $\mathcal{S}$ of plan and term triples with the following properties: (1) for every $\langle S, src, tgt\rangle \in \mathcal{S}$, $[\![S]\!]$ is sound with respect to $\mathcal{A}$, and (2) for every shortcut path p from src to tgt in a solution graph for $t \in T$, $\mathcal{A}$, and* Reduced*, there is a triple $\langle S, src, tgt\rangle \in \mathcal{S}$ such that S replays p.*

4.2 Rule Synthesis

RuleSy synthesizes tactics by searching for well-formed programs that satisfy specifications $\langle S, src, tgt\rangle$ produced by FindSpecs. This search is a form of syntax-guided synthesis [13]: it draws candidate programs from a given syntactic space, and uses an automatic verifier to check if a chosen candidate satisfies the specification. We illustrate the challenges of classic syntax-guided synthesis for rule programs; show how our *best-implements* query addresses them; and present the FindRules algorithm for sound, complete, and efficient solving of this query.

Classic Synthesis for Rule Programs. In our setting, the classic synthesis query takes the form $\exists R. \forall t. [\![R]\!]t = [\![S]\!]t$, where R is a well-formed program and S is an execution plan. Existing tools [13,11,12] cannot solve this query soundly because it involves verifying candidate programs over terms of unbounded size.

But even if we weaken the soundness guarantee to functional correctness over bounded inputs, these tools can fail to find useful rules because the classic

```
1: function FindRules(S: plan, src, tgt: terms, k̄: ints)
2:   idx ← replayIndex(S, src, tgt)
3:   s, t ← ref(src, idx), [[S]]ref(src, idx)
4:   p0 ← termToPattern(s)                          ▷ Most refined pattern that matches s
5:   R ← ⋃_{p0 ⊑ p} FindRule(p, S, s, t, k̄)
6:   return R                                       ▷ Rules that best implement S for ⟨src, tgt⟩

7: function FindRule(p: pattern, S: plan, s,t: terms, k̄: ints) ▷ [[p]]s ∧ t=[[S]]s
8:   ??c ← WellFormedConstraintHole(p, k̄)
9:   C ← (Condition (Pattern p) (Constraint ??c))   ▷ Condition sketch
10:  ??a‾ ← WellFormedCommandHoles(p, k̄)
11:  A ← (Action ??a‾)                ▷ Action sketch with a sequence ??a‾ of holes
12:  𝕋 ← {t | [[p]]t}                 ▷ Symbolic representation of all terms that satisfy p
13:  c ← CEGIS([[C]]s ∧ (∀τ ∈ 𝕋.[[C]]τ ⟺ [[S]]τ ≠ ⊥))
14:  a ← CEGIS([[A]]s = t ∧ (∀τ ∈ 𝕋.[[S]]τ ≠ ⊥ ⟹ [[A]]τ = [[S]]τ))
15:  return {(Rule c a) | c ≠ ⊥ ∧ a ≠ ⊥}
```

Fig. 8: FindRules takes as input a bound $\bar{k}$ on program size and an execution plan S that replays a path from *src* to *tgt*. Given these inputs, it synthesizes all rule programs of size $\bar{k}$ that best implement S with respect to *src* and *tgt*.

query is overly strict for our purposes. To see why, consider the specification $\langle S, src, tgt\rangle$ where S is $[\langle \mathbf{A}, [1], \beta_0\rangle, \langle \mathbf{A}, [2], \beta_0\rangle]$, *src* is $(+\,(+\,0\,x)\,(+\,0\,y))$, *tgt* is $(+\,x\,y)$, $\mathbf{A}$ is the additive identity axiom (Figure 1a), and β_0 is the identity binding. The plan S specifies a general tactic for transforming a term of the form $(op\,(+\,0\,e_0)\,(+\,0\,e_1))$ to the term $(op\,e_0\,e_1)$, where op is any binary operator in our algebra DSL. Such a tactic cannot be expressed as a well-formed program (Definition 5). But many useful specializations of this tactic are expressible, e.g.:

```
(Rule (Condition
        (Pattern (Term + (Term + (ConstTerm) _) (Term + (ConstTerm) _) etc))
        (Constraint (And (Eq? (Ref 1 1) 0) (Eq? (Ref 2 1) 0))))
      (Action (Replace (Ref 1) (Ref 1 2)) (Replace (Ref 2) (Ref 2 2))))
```

Since we aim to generate useful tactics for domain model optimization, an ideal synthesis query for RuleSy would admit many such specialized yet widely applicable implementations of S.

The Best-Implements Synthesis Query. To address the challenges of classic synthesis, we reformulate the synthesis task for RuleSy as follows: given $\langle S, src, tgt\rangle$, find *all* rules R that fire on *src* to produce *tgt*, that are sound with respect to S, and that capture a locally maximal subset of the behaviors specified by S. We say that such rules *best implement* S for $\langle src, tgt\rangle$ (Definition 12), and we search for them with the FindRules algorithm (Figure 8), which is a sound and complete synthesis procedure for the best-implements query (Theorem 2).

Definition 12 (Best Implementation). *Let S be an execution plan that replays a path from a term src to a term tgt. A well-formed rule R* best implements *S for $\langle src, tgt\rangle$ if $tgt \in fire(R, src)$ and $\forall t. [\![pattern(R)]\!]t \implies [\![R]\!]t = [\![S]\!]t$.*

Sound and Complete Verification. Verifying that a program R best implements a plan S involves checking that R produces the same output as S on all terms t accepted by R's pattern. The verification task is therefore to decide the validity of the formula $\forall t. [\![pattern(R)]\!]t \implies [\![R]\!]t = [\![S]\!]t$. We do so by observing [17] that

this formula has a small model property when R is well-formed (Definition 5): if the formula is valid on a carefully constructed finite set of terms $\mathbb{T}$, then it is valid on all terms. At a high level, $\mathbb{T}$ consists of terms that satisfy R's pattern in a representative fashion. For example, $\mathbb{T} = \{x\}$ for the pattern `(VarTerm)` because all terms that satisfy `(VarTerm)` are isomorphic to the variable x up to a renaming. Encoding the set $\mathbb{T}$ symbolically (rather than explicitly) enables FindRules to discharge its verification task efficiently with an off-the-shelf SMT solver [18].

Efficient Search. FindRules accelerates synthesis by exploiting the observation that a best implementation of $\langle S, src, tgt\rangle$ must fire on *src* to produce *tgt*, which has two key consequences. First, because S replays a path from *src* to *tgt* (Theorem 1), *src* contains a subterm s at an index *idx* such that $t = [\![S]\!]s$ and $tgt = replace(src, idx, t)$ (lines 2-3). Any rule R that outputs t on s will therefore fire on *src* to produce *tgt*, so it sufficient to look for rules R that transform s to t, without having to reason about the semantics of *fire*. Second, if a rule accepts s, its pattern must be refined (Definition 13) by the most specific pattern p_0 (line 4) that accepts s. To construct p_0, we replace each literal in s with `(ConstTerm)`, variable with `(VarTerm)`, and operator o with the tokens `Term` o. Since p_0 refines finitely many patterns p, we can enumerate all of them (line 5). Once p is fixed through enumeration, FindRule can efficiently search for a best implementation R with that pattern, by using an off-the-shelf synthesizer [12] to perform two independent searches for R's condition (line 13) and action (line 14). These two searches explore an exponentially smaller candidate space than a single search for the condition and action [17], without missing any correct rules (Theorem 2).

Definition 13 (Pattern Refinement). *A condition pattern p_1* refines *a pattern p_2 if $p_1 \sqsubseteq p_2$, where $\sqsubseteq$ is defined as follows: $p \sqsubseteq p$; $p \sqsubseteq$ _; `(ConstTerm)` $\sqsubseteq$ `(BaseTerm)`; `(VarTerm)` $\sqsubseteq$ `(BaseTerm)`; `(Term` $o\ p_1 \ldots p_k$`)` $\sqsubseteq$ `(Term` $o\ q_1 \ldots q_k$`)` if $p_i \sqsubseteq q_i$ for all $i \in [1..k]$; and `(Term` $o\ p_1 \ldots p_n$`)` $\sqsubseteq$ `(Term` $o\ q_1 \ldots q_k$ `etc)` if $n \geq k$ and $p_i \sqsubseteq q_i$ for all $i \in [1..k]$.*

Theorem 2. *Let S be an execution plan that replays a shortcut path from src to tgt, and $\bar{k}$ a bound on the size of rule programs.* FindRules$(S, src, tgt, \bar{k})$ *returns a set of rules $\mathcal{R}$ with the following properties: (1) every $R \in \mathcal{R}$ best implements S for $\langle src, tgt\rangle$; (2) $\mathcal{R}$ includes a sound rule R of size $\bar{k}$ if one exists; and (3) for every pattern p that refines or is refined by R's pattern, $\mathcal{R}$ includes a sound rule with pattern p and size $\bar{k}$ if one exists.*

4.3 Rule Set Optimization

After synthesizing the tactics $\mathcal{T}$ for the examples T and axioms $\mathcal{A}$, RuleSy applies discrete optimization to find a subset of $\mathcal{A} \cup \mathcal{T}$ that minimizes the objective function f. We formulate this optimization problem in a way that guarantees termination. In particular, our Optimize algorithm (Figure 9) returns a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that can solve each example in T and that minimize f over all *shortest* solution graphs for T and $\mathcal{A} \cup \mathcal{T}$ (Theorem 3). Restricting the optimization to shortest solutions enables us to decide whether an arbitrary rule set

```
 1: function Optimize(T: set of terms, A,T: set of rules, f: objective)
 2:   G_{A∪T} ← {}
 3:   for t ∈ T such that ¬Reduced(t) do
 4:     ⟨N, E_A⟩ ← Solve(t, A)                                   ▷ Solve with axioms
 5:     E_T ← ⋃_{R∈T} ⋃_{s,t∈N} {⟨s, t⟩ | t ∈ fire(R, s)}          ▷ Tactic edges
 6:     G_{A∪T} ← G_{A∪T} ∪ {⟨N, E_A ∪ E_T⟩}
 7:   f_∅ ← λR.G. if ⟨∅, ∅⟩ ∈ G then return ∞ else return f(R, G)
 8:   return min_{R⊆A∪T} f_∅(R, {Restrict(G, R) | G ∈ G_{A∪T}})
 9: function Restrict(⟨N, E⟩: solution graph, R: set of rules)
10:   t ← source of the graph ⟨N, E⟩
11:   E_R ← {⟨src, tgt⟩_R ∈ E | R ∈ R}                            ▷ Edges with labels in R
12:   paths ← ⋃_{t̂∈N ∧ Reduced(t̂)} allPaths(t, t̂, ⟨N, E_R⟩)
13:   E ← ⋃_{p∈paths} pathEdges(p)
14:   N ← {n | ∃e ∈ E . source(e) = n ∨ target(e) = n}
15:   return ⟨N, E⟩                          ▷ Solution graph for t and R or ⟨∅, ∅⟩
```

Fig. 9: Optimize takes as input a set of terms T, axioms $\mathcal{A}$ for reducing T, tactics $\mathcal{T}$ synthesized from $\mathcal{A}$ and T using FindRules and FindSpecs, and an objective function f. The output is a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that minimizes f.

$\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ can solve an example $t \in T$ without having to invoke Solve$(t, \mathcal{R})$, which may not terminate for an arbitrary term t and rule set $\mathcal{R}$ in our DSL.

The Optimize procedure works in three steps. First, for each example term $t \in T$, lines 4-5 construct a solution graph $\langle N, E_{\mathcal{A}} \cup E_{\mathcal{T}} \rangle$ that contains shortest solutions for t and all subsets of $\mathcal{A} \cup \mathcal{T}$. Next, line 7 creates a function $f_{\emptyset}$ that takes as input a set of rules $\mathcal{R}$ and a set of graphs $\mathcal{G}$, and produces ∞ if $\mathcal{G}$ contains the empty graph (indicating that $\mathcal{R}$ cannot solve some term in T) and $f(\mathcal{R}, \mathcal{G})$ otherwise. Finally, line 8 searches for $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that minimizes f over $\mathcal{G}_{\mathcal{A}\cup\mathcal{T}}$. This search relies on the procedure Restrict$(G, \mathcal{R})$ to extract from G a solution graph for $t \in T$ and $\mathcal{R}$ if one is included, or the empty graph otherwise. For linear objectives f, the search can be delegated to an optimizing SMT solver [18]. For other objectives (e.g., Figure 1c), we use a greedy algorithm to find a locally minimal solution (thus weakening the optimality guarantee in Theorem 3).

Theorem 3. *Let $\mathcal{T}$ be a set of tactics synthesized by* RuleSy *for terms T and axioms $\mathcal{A}$, and let f be a total function from sets of rules and solution graphs to positive real numbers.* Optimize$(T, \mathcal{A}, \mathcal{T}, f)$ *returns a set of rules $\mathcal{R} \subseteq \mathcal{A} \cup \mathcal{T}$ that can solve each term in T, and for all such $\mathcal{R}' \subseteq \mathcal{A} \cup \mathcal{T}$, $f(\mathcal{R}, \{\text{Solve}(t, \mathcal{R}) \mid t \in T\}) \leq f(\mathcal{R}', \{\text{Solve}(t, \mathcal{R}') \mid t \in T\})$.*

5 Evaluation

To evaluate RuleSy's effectiveness at synthesizing domain models, we answer the following four research questions:

RQ 1. Can RuleSy's synthesis algorithm recover standard tactics from a textbook and discover new ones?

RQ 2. Can RuleSy's optimization algorithm recover textbook domain models and discover variants of those models that optimize different objectives?

RQ 3. Does RULESY significantly outperform RULESYNTH, a prior tool [15] for modeling the domain of introductory K-12 algebra?
RQ 4. Can RULESY support different educational domains?

The first two questions assess the quality of RULESY's output by comparing the synthesized tactics and domain models to a textbook [9] written by domain experts. The third question evaluates the performance of RULESY's algorithms by comparison to an existing tool for synthesizing tactics and domain models. The fourth question assesses the generality of our approach. We conducted two case studies to answer these questions, finding positive answers to each. The implementation source code and evaluation data are available online [19].

5.1 Case Study with Algebra (RQ 1–3)

We performed three experiments in the domain of K-12 algebra to answer RQ 1–3. Each experiment was executed on an Intel 2$^{\text{nd}}$ generation i7 processor with 8 virtual threads. The system was limited to a synthesis timeout of 20 minutes per mined specification. The details and results are presented below.

Table 1: Example problems (a) and axioms (b) for the algebra case study.

(a) Example problems.

ID	Source	#
P_R	RULESYNTH [15]	55
P_T	Chapter 2, Sections 1-4 of Hall et al. [9]	92

(b) Axioms.

ID	Name	Example
A	Additive Identity	$x + 0 \to x$
B	Adding Constants	$2 + 3 \to 5$
C	Multiplicative Identity	$1x \to x$
D	Multiplying by Zero	$0(x + 2) \to 0$
E	Multiplying Constants	$2 * 3 \to 6$
F	Divisive Identity	$\frac{x}{1} \to x$
G	Canceling Fractions	$\frac{2x}{2y} \to \frac{x}{y}$
H	Multiplying Fractions	$3\left(\frac{2x}{4}\right) \to \frac{(2*3)x}{4}$
I	Factoring	$3x + 4x \to (3 + 4)x$
J	Distribution	$(3 + 4)x \to 3x + 4x$
K	Expanding Terms	$x \to 1x$
L	Expanding Negatives	$-x \to -1x$
M	Adding to Both Sides	$x + -1 = 2 \to x + -1 + 1 = 2 + 1$
N	Dividing Both Sides	$3x = 2 \to \frac{3x}{3} = \frac{2}{3}$
O	Multiplying Both Sides	$\frac{x}{3} = 2 \to 3\left(\frac{x}{3}\right) = 2 * 3$

Quality of Synthesized Rules (RQ 1). To evaluate the quality of the rules synthesized by RULESY, we apply the system to the examples (P_T in Table 1a) and axioms (Table 1b) from a standard algebra textbook [9], and compare system output (607 tactics) to the tactics from the textbook. Since the book demonstrates rules on examples rather than explicitly, determining which rules are shown involves some interpretation. For example, we interpret the transformation $5x+2-2x = 2x+14-2x \to 3x+2 = 14$ as demonstrating two independent tactics, one for each side of the equation, rather than one tactic with unrelated subparts. The second column of Table 2 lists all the tactics presented in the book. We find that RULESY recovers each of them or a close variation.

In addition to recovering textbook tactics, RuleSy finds interesting variations on rules commonly taught in algebra class. Figure 10 shows an example, which isolates a variable from a negated fraction and an addend. This rule composes 9 axioms, demonstrating RuleSy's ability to discover advanced tactics.

```
(define MBALNGOHG ; Isolate a variable from a negated fraction and an addend:
  (Rule           ; (= (+ (- (/ (* x ...) b)) c) e) → (= (* x ...) (* b (- c e)))
    (Condition
      (Pattern
        (Term = (Term + (Term - (Term / (Term * (VarTerm) etc) (BaseTerm)))
                        (ConstTerm))
                _))
      (Constraint true))
    (Action
      (Replace (Ref 1) (Ref 1 1 1 1))
      (Replace (Ref 2) (Make * (Ref 1 1 1 2) (Make - (Ref 1 2) (Ref 2)))))))
```

Fig. 10: A custom algebra tactic discovered by RuleSy.

```
(define xpq ; Modus ponens: if I ⊨ A → B and I ⊨ A, then I ⊨ B.
  (Rule
    (Condition (Pattern (Term known (Term ⊨ (Term → _ _)) (Term ⊨ _) etc))
               (Constraint (Eq? (Ref 1 1 1) (Ref 2 1))))
    (Action (Replace (Ref) (Cons (Make ⊨ (Ref 1 1 2)) (Ref))))))
```

Fig. 11: A proof tactic synthesized by RuleSy.

Quality of Synthesized Domain Models (RQ 2). We next evaluate RuleSy's ability to recover textbook domain models along with variations that optimize different objectives. An important part of creating domain models for educational tools (and curricula in general) is choosing the *progression*—the sequence in which different concepts (i.e., rules) should be learned. We use RuleSy and the objective function shown in Figure 1c to find a progression of optimal domain models for the problems (P_T in Table 1a) and axioms (Table 1b) in [9], and we compare this progression to the one in the book.

We create a progression by producing a sequence of domain models for Sections 1–4 of Chapter 2 in [9]. Every successive model is constrained to be a superset of the previous model(s): students keep what they learned and use it in subsequent sections. To generate a domain model D_n for section n, we apply RuleSy's optimizer to the exercise problems from section n; the objective function in Figure 1c with $\alpha \in \{.05, .125, .25\}$; and all available rules (axioms and tactics), coupled with the constraint that $D_1 \cup \ldots \cup D_{n-1} \subseteq D_n$.

Table 2 shows the resulting progressions of optimal domain models for [9], along with the rules that are introduced in the corresponding sections. For each rule presented in a section, the corresponding optimal model for $\alpha = .05$ contains either the rule itself or a close variation. Increasing α leads to new domain models that emphasize rule set complexity over solution efficiency. This result

demonstrates that RuleSy can recover textbook domain models, as well as find new models that optimize different objectives.

Table 2: A textbook [9] progression and the corresponding optimal domain models found by RuleSy, using 3 settings of α (Figure 1c). Row i shows the rules that the ith model adds to the preceding models.

Section	Textbook Rules	ODM $\alpha = 0.05$	ODM $\alpha = 0.125$	ODM $\alpha = 0.25$
2-1	**B**, **M**, **N**, **G**, **O**, **BA**, **HG**	**M**, **A**, **K**, **L**, **LNG**, **NG**, **OHG**, **IBD**, **MBA**	**NG**, **OHG**, **MBA**	**NG**, **OHG**, **MBA**
2-2	**L**, **E**	**LE**	**LNG**, **LE**	**E**, **L**
2-3	**J**, **IB**, **KIB**, **JB**	**E**, **J**, **KIB**, **IB**, **BMBA**	**E**, **K**, **L**, **J**, **B**, **IB**	**I**, **K**, **J**, **B**
2-4	**LEIBDA**, **LEIB**	**C**, **BD**, **LEIB**, **MLEI**	**M**, **C**, **BD**, **IBD**, **LEIB**, **MLEI**	**M**, **C**, **D**, **LEIB**

Comparison to Prior Work (RQ 3). We compare the performance of RuleSy to the prior system RuleSynth by applying both tools to the example problems P_R in Table 1a and the axioms in Table 1b. We use the same problems as the original evaluation of RuleSynth because its algorithms encounter performance problems on the (larger) textbook problems P_T. Given these inputs, RuleSy synthesizes 144 tactics, which include the 13 rules synthesized by RuleSynth. Figure 12 graphs the rate of rules produced by each system, which accounts for the time to mine specifications and synthesize rules for those specifications. Our system both learns more rules and does so at a faster rate.

RuleSy outperforms RuleSynth thanks to the soundness and completeness of its specification mining and synthesis algorithms. RuleSynth employs a heuristic four-step procedure for synthesizing tactics: (1) use the axioms to solve the example problems; (2) extract pairs of input-output terms for all axiom sequences that appear in the solutions; (3) heuristically group those pairs into sets that are likely to be specifying the same tactics; and (4) synthesize a tactic for each resulting set. This process is neither sound nor complete, so RuleSynth can produce incorrect tactics and miss tactic specifications found by RuleSy.

To show that RuleSy can efficiently explore spaces of rules to find optimal domain models, we compare its runtime performance to that of RuleSynth. Since the two systems use different input languages, we manually transcribed the 13 tactics generated by RuleSynth into our algebra DSL. Given these tactics, the axioms in Table 1b, and the examples P_R, RuleSynth finds an optimal rule set in 20 seconds, whereas RuleSy takes 14 seconds. As the optimization is superlinear in the number of rules, we can expect this performance difference to be magnified on larger rule sets. Figure 13 shows that RuleSy's optimization algorithm finds domain models quickly, even on much larger design spaces.

5.2 Case Study with Propositional Logic (RQ 4)

To evaluate the extensibility and generality of RuleSy, we applied it to the domain of semantic proofs for elementary propositional logic theorems. Many students have trouble learning how to construct proofs [20], so custom educational tools could help by teaching a variety of proof strategies.

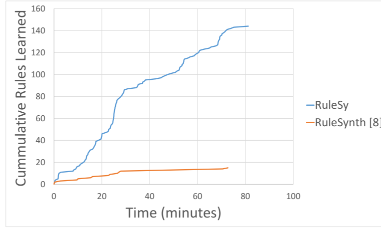


Fig. 12: The number of rules synthesized by RuleSy (blue) and RuleSynth (orange) over time on the same inputs. RuleSy learns 10× more rules at a quicker rate.

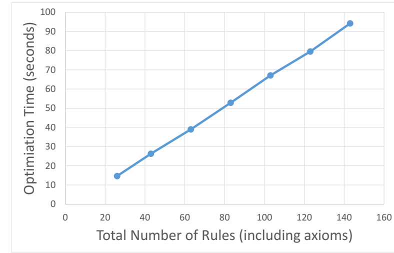


Fig. 13: Optimizer performance on design spaces derived from P_R of various sizes, using the objective in Figure 1c with $\alpha = .125$.

Table 3: The axioms [16] used for the logic case study.

ID	Name	Description
p	Contradiction	If $I \vDash A$ and $I \nvDash A$ then $I \vDash \bot$
q	Branch elimination	If $I \vDash \bot \mid A$ then $I \vDash A$
r	And 1	If $I \vDash A \wedge _$ then $I \vDash A$
s	And 2	If $I \nvDash A \wedge B$ then $I \nvDash A \mid I \nvDash B$
t	Or 1	If $I \vDash A \vee B$ then $I \vDash A \mid I \vDash B$
u	Or 2	If $I \nvDash A \vee _$ then $I \nvDash A$
v	Not 1	If $I \vDash \neg A$ then $I \nvDash A$
w	Not 2	If $I \nvDash \neg A$ then $I \vDash A$
x	Implication 1	If $I \vDash A \rightarrow B$, then $I \nvDash A \mid I \vDash B$
y	Implication 2	If $I \nvDash A \rightarrow B$, then $I \vDash A$
z	Implication 3	If $I \nvDash A \rightarrow B$, then $I \nvDash B$

We instantiated RuleSy with a DSL for expressing semantic proofs. The DSL represents problem states as proof trees, consisting of a set of branches, each containing a set of facts that have been proven so far. The DSL encodes this proof structure with commutative operators `branch` and `known`. The problem-solving task in this domain is to establish the validity of a propositional formula, such as $(p \wedge q) \rightarrow (p \rightarrow q)$, by assuming a falsifying interpretation and applying proof rules to arrive at a contradiction in every branch. Tactics apply multiple proof steps (i.e., axioms) at once.

We applied this instantiation of RuleSy to the axioms (Table 3) and proof exercises (3 in total) from a textbook [16]. The system synthesized 85 rules in 72 minutes. The resulting rules includes interesting general proof rules for each of the exercises. For example, given $(p \wedge (p \rightarrow q)) \rightarrow q$, RuleSy mines and synthesizes the modus ponens tactic shown in Figure 11. These results show RuleSy's applicability and effectiveness extend beyond the domain of K-12 algebra.

6 Related Work

Automated Rule Learning. Automated rule learning is a well-studied problem in Artificial Intelligence and Machine Learning. RuleSy is most closely related to

rule learning approaches in discrete planning domains, such as cognitive architectures [21]. Its learning of tactics from axioms is similar to chunking in SOAR [22], knowledge compilation in ACT [23], and macro-learning from AI planning [24]. But unlike these systems, RuleSy can learn rules for transforming problems represented as trees, and express objective criteria over rules and solutions.

Inductive Logic Programming. Within educational technology, researchers have investigated automated learning of rules and domain models for intelligent tutors [25]. Previous efforts have focused on applying inductive logic programming to learn a domain model from a set of expert solution traces [26,27,28,29,30]. RuleSy, in contrast, uses a small set of axioms and example problems to synthesize an exhaustive set of sound tactics, and it searches the axioms and tactics for a model that optimizes a desired objective.

Program Synthesis. Prior educational applications of program synthesis and automated search include problem and solution generation [31,1], hint and feedback generation [32,33,34], and checking of student proofs [35]. RuleSy solves a different problem: generating condition-action rules and domain models. General approaches to programming-by-example [36,37] have investigated the problem of learning useful programs from a small number of input-output examples, with no general soundness guarantees. RuleSy, in contrast, uses axioms to verify that the synthesized programs are sound for all inputs, relying on examples only to bias the search toward useful programs (i.e., tactics that shorten solutions).

Term Rewrite Systems. RuleSy helps automate the construction of rule-based domain models, which are related to term rewrite systems [38]. Our work can be seen as an approach for learning rewrite rules, and selecting a cheapest rewrite system that terminates on a given finite set of terms. RuleSy terms are a special case of recursive data types, which have been extensively studied in the context of automated reasoning [39,40,41]. Our rule language is designed to support effective automated reasoning by reduction to the quantifier-free theory of bitvectors.

7 Conclusion

This paper presented RuleSy, a framework for computer-aided development of domain models expressed as condition-action rules. RuleSy is based on new algorithms for mining specifications of tactic rules from examples and axioms, synthesizing sound implementations of those specifications, and selecting an optimal domain model from a set of axioms and tactics. Thanks to these algorithms, RuleSy efficiently recovers textbook tactic rules and models for K-12 algebra, discovers new ones, and generalizes to other domains. As the need for tools to support personalized education grows, RuleSy can help tool developers rapidly create domain models that target individual students' educational objectives.

Acknowledgements. This research was supported in part by NSF CCF-1651225, 1639576, and 1546510; Oak Foundation 16-644; and Hewlett Foundation.

References

1. Andersen, E., Gulwani, S., Popović, Z.: A trace-based framework for analyzing and synthesizing educational progressions. In: CHI. (2013)
2. O'Rourke, E., Andersen, E., Gulwani, S., Popović, Z.: A framework for automatically generating interactive instructional scaffolding. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems. CHI '15, New York, NY, USA, ACM (2015) 1545–1554
3. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: Lessons learned. The journal of the learning sciences **4**(2) (1995) 167–207
4. VanLehn, K.: Mind bugs: The origins of procedural misconceptions. MIT press (1990)
5. Murray, T.: Authoring intelligent tutoring systems: An analysis of the state of the art. International Journal of Artificial Intelligence in Education **10** (1999)
6. Liu, Y.E., Ballweber, C., O'rourke, E., Butler, E., Thummaphan, P., Popović, Z.: Large-scale educational campaigns. ACM Trans. Comput.-Hum. Interact. **22**(2) (March 2015) 8:1–8:24
7. Demski, J.: This time it's personal: True student-centered learning has a lot of support from education leaders, but it can't really happen without all the right technology infrastructure to drive it. and the technology just may be ready to deliver on its promise. THE Journal (Technological Horizons In Education) **39**(1) (2012) 32
8. Redding, S.: Getting personal: The promise of personalized learning. Handbook on innovations in learning (2013) 113–130
9. Charles, R.I., Hall, B., Kennedy, D., Bellman, A.E., Bragg, S.C., Handlin, W.G., Murphy, S.J., Wiggins, G.: Algebra 1: Common Core. Pearson Education, Inc. (2012)
10. Sweller, J.: Cognitive load during problem solving: Effects on learning. Cognitive science **12**(2) (1988) 257–285
11. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th Inter. Conf. on Architectural Support for Programming Languages and Operating Systems, ACM (2006)
12. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. (2014)
13. Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M.M.K., Raghothaman, M., Saha, S., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Dependable Software Systems Engineering. (2015) 1–25
14. Dershowitz, N., Jouannaud, J.P.: Handbook of theoretical computer science (vol. b). MIT Press, Cambridge, MA, USA (1990) 243–320
15. Butler, E., Torlak, E., Popović, Z.: A framework for parameterized design of rule systems applied to algebra. In: Intelligent Tutoring Systems, Springer (2016)
16. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
17. Butler, E., Torlak, E., Popovic, Z.: Synthesizing optimal domain models for educational applications. Technical Report UW-CSE-17-10-02, https://www.cs.washington.edu/tr/2017/10/UW-CSE-17-10-02.pdf, University of Washington (2017)

18. Moura, L., Bjørner, N.: 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. Springer Berlin Heidelberg, Berlin, Heidelberg (2008) 337–340
19. Butler, E., Torlak, E., Popovic, Z.: Rulesy source code and data. `https://github.com/edbutler/nonograms-rule-synthesis`
20. Harel, G., Sowder, L.: Toward comprehensive perspectives on the learning and teaching of proof. Second handbook of research on mathematics teaching and learning **2** (2007) 805–842
21. Langley, P., Laird, J.E., Rogers, S.: Cognitive architectures: Research issues and challenges. Cognitive Systems Research **10**(2) (2009) 141–160
22. Laird, J.E., Newell, A., Rosenbloom, P.S.: Soar: An architecture for general intelligence. Artificial Intelligence **33**(1) (1987) 1 – 64
23. Anderson, J.R., Lebiere, C.: The atomic components of thought. (1998)
24. Korf, R.E.: Macro-operators: A weak method for learning. Artificial Intelligence **26**(1) (1985) 35 – 77
25. Koedinger, K.R., Brunskill, E., Baker, R.S., McLaughlin, E.A., Stamper, J.: New potentials for data-driven intelligent tutoring system development and optimization. AI Magazine **34**(3) (2013) 27–41
26. Matsuda, N., Cohen, W.W., Koedinger, K.R.: Applying programming by demonstration in an intelligent authoring tool for cognitive tutors. In: Aaai workshop on human comprehensible machine learning (technical report ws-05-04). (2005) 1–8
27. Li, N., Cohen, W., Koedinger, K.R., Matsuda, N.: A machine learning approach for automatic student model discovery. In: Educational Data Mining 2011. (2010)
28. Li, N., Schreiber, A.J., Cohen, W., Koedinger, K.: Efficient complex skill acquisition through representation learning. Advances in Cognitive Systems **2** (2012)
29. Jarvis, M.P., Nuzzo-Jones, G., Heffernan, N.T.: Applying machine learning techniques to rule generation in intelligent tutoring systems. In: Intelligent Tutoring Systems, Springer (2004) 541–553
30. Schmid, U., Kitzelmann, E.: Inductive rule learning on the knowledge level. Cognitive Systems Research **12**(3) (2011) 237–248
31. Gulwani, S.: Example-based learning in computer-aided stem education. Communications of the ACM **57**(8) (2014) 70–80
32. Lazar, T., Bratko, I.: Data-driven program synthesis for hint generation in programming tutors. In: Intelligent Tutoring Systems, Springer (2014) 306–311
33. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. (2013)
34. Tillmann, N., de Halleux, J., Xie, T., Bishop, J.: Constructing coding duels in Pex4Fun and Code Hunt. In: ISSTA, ACM (2014) 445–448
35. Lee, C.: DeduceIt: a tool for representing and evaluating student derivations. Stanford Digital Repository: http://purl.stanford.edu/bg823wn2892 (2012)
36. Polozov, O., Gulwani, S.: Flashmeta: A framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN Inter. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, ACM (2015) 107–126
37. Liang, P., Jordan, M.I., Klein, D.: Learning programs: A hierarchical bayesian approach. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10). (2010) 639–646
38. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. Citeseer (1989)

39. Oppen, D.C.: Reasoning about recursively defined data structures. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM (1978) 151–157
40. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. Acm Sigplan Notices **45**(1) (2010) 199–210
41. 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) (2007) 23–37

Automatic Verification of Intermittent Systems

Manjeet Dahiya and Sorav Bansal

Indian Institute of Technology Delhi
{dahiya, sbansal}@cse.iitd.ac.in

Abstract. Transiently powered devices have given rise to a new model of computation called *intermittent computation*. Intermittent programs keep checkpointing the program state to a persistent memory, and on power failures, the programs resume from the last executed checkpoint. An intermittent program is usually automatically generated by instrumenting a given continuous program (continuously powered). The behaviour of the continuous program should be equivalent to that of the intermittent program under all possible power failures.
This paper presents a technique to automatically verify the correctness of an intermittent program with respect to its continuous counterpart. We present a model of intermittence to capture all possible scenarios of power failures and an algorithm to automatically find a proof of equivalence between a continuous and an intermittent program.

1 Introduction

Energy harvesting devices, that harvest energy from their surroundings, such as sunlight or RF radio signals, are increasingly getting popular. Because the size reduction of batteries has not kept pace with the size reduction of transistor technology, energy harvesting allows such devices to be much smaller in size, e.g., insect-scale wildlife tracking devices [19] and implantable medical devices [17]. Such devices are already commonplace for small dedicated computations, e.g., challenge-response in passive RFID cards, and are now being imagined for more general-purpose computational tasks [15, 19].

The harvested energy is unpredictable and usually not adequate for continuous operation of a device. Power failures are spontaneous, and may occur after every 100 milliseconds, for example [19]. Thus, computation needs to be split into small chunks that can finish in these small intervals of operation, and intermediate results need to be saved to a persistent memory device at the end of each interval. A power reboot should then be able to resume from the results of the last saved computational state. This model of computation has also been termed, *intermittent computation* [15]. Typically, the intermittent programs involve instrumentation of the continuous programs (that are supposed to be continuously powered) with periodic *checkpoints*. The checkpoints need to be close enough, so that the computation across two checkpoints can finish within one power cycle. On the other hand, frequent checkpoints degrade efficiency during continuous operation. Further, a checkpoint need not save all program state, but can save

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 161–182, 2018.
https://doi.org/10.1007/978-3-319-73721-8_8

only the *necessary* program state elements, required for an acceptable computational state at reboot. The presence of volatile and non-volatile program state simultaneously, makes the problem more interesting.

An intermittent program may be written by hand, through manual reasoning. Alternatively, semi-automatic [15] and automatic [19,26] tools can be used to instrument continuous programs with checkpoints, to allow them to execute correctly in the intermittent environments. The goal of these automated tools is to generate an intermittent program that is equivalent to the continuous program under all possible power failures. In addition to correctness, these tools try to generate intermittent programs with smaller checkpoints for efficiency. These tools reason over high-level programs (C or LLVM IR), and it has been reported that it is a challenge [26] to work at a higher level. Given that the failures happen at the architecture instruction granularity (and possibly at microinstruction granularity) and it is the machine state that needs to be checkpointed; the reasoning at a higher level is error-prone and could go wrong because of the transformations (e.g., instruction reordering) performed by the compiler. Moreover, the bugs in intermittent programs could be very hard to detect because the number of states involved is very large due to spontaneous and recurring power failures.

Verifying the correctness of an intermittent program with respect to a continuous program is important from two aspects: First, we will be able to verify the correctness of the output of existing automatic instrumentation tools. Second, a verification tool will enable us to model automatic-instrumentation as a synthesis problem to optimize for the efficiency of generated intermittent programs, with the added confidence of verified output.

We present an automatic technique to verify the correctness of an intermittent program with respect to a continuous program. Towards this goal, we make the following contributions: (a) A formal model of *intermittence* that correctly and exhaustively captures the behaviour of intermittent programs for all possible power failures. Additionally, the model of intermittent programs is amenable to checking equivalence with its continuous counterpart. (b) Due to recurring executions in an intermittent program, an intermediate observable event (not occurring at exit) may occur multiple times, causing an equivalence failure. We show that if the observables are *idempotent* and *commutative*, then we can claim equivalence between the two programs. (c) A robust algorithm to infer a provable bisimulation relation to establish equivalence across a continuous and an intermittent program. The problem is undecidable in general. The algorithm is robust in the sense of its generality in handling even minimal checkpointing states, i.e, a more robust algorithm can verify an intermittent program with smaller checkpoints. In other words, we perform *translation validation* of the translation from continuous to intermittent programs. However, in our case, in addition to program transformation, the program execution environment also changes. The continuous program is supplied with continuous power whereas the intermittent program is powered with transient power.

We have implemented our algorithm in a tool and evaluated it for verification runtime and robustness. For measuring the robustness, we implemented a synthesis loop to greedily minimize the checkpointed state elements at a given set of checkpoint locations. The synthesis loop proposes smaller checkpoints, and relies on our equivalence procedure for checking equivalence between the continuous and the intermittent program, under the proposed checkpoints. The synthesis loop can result in a smaller checkpoint if our equivalence procedure can verify the same, i.e., optimization is dependent on the robustness of our verification algorithm. We tested our tool on the benchmarks from the previous work and compared our results with DINO [15]. The synthesis loop is able to produce checkpoints whose size is on average 4 times smaller than that of the checkpoints produced by DINO. The synthesis time ranges from 42 secs to 7 hours, and the average verification time is 73 secs.

2 Example

We briefly discuss, with the help of an example, the working of intermittent programs and issues associated with it. Fig. 1a shows an x86 program that increments a *non-volatile* global variable `nv` and returns `0` on success. The program terminates after returning from this procedure. We call it a continuous program as it is not meant to work in an environment with power failures. Fig. 1b shows an intermittent program, generated by instrumenting the continuous program. This program can tolerate power failures, and it is equivalent to the continuous program, under all possible power failures. The equivalence is computed with respect to the observable behaviour, which in this case is the output, i.e., the value of return register `eax` and the value of the global variable `nv`.

The intermittent program has been generated from the continuous program by inserting checkpointing logic at the checkpoint locations `CP1` and `CP2`. During checkpointing, the specified `CPelems` and the location of the current executing checkpoint get saved to `CPdata` in persistent memory. In case of a power failure, the program runs from the entry again, i.e., the restoration logic, it restores the `CPelems`, and then jumps to the location stored in `CPdata.eip`. For the first run of the intermittent program, the checkpoint data is initialized to `((), Entry)`, i.e., `CPdata.CPelems=() and CPdata.eip=Entry`. This ensures that on the first run, the restoration logic takes the program control flow to the original entry of the program. More details on instrumentation are discussed in Sec. 4.1.

In case of power failures, the periodic checkpointing allows the intermittent programs to not lose the computation and instead, start from the last executed checkpoint. For example, if a failure occurs at location `I5`, the intermittent program will resume its computation correctly from `CP2`, on power reboot. This is so because the checkpoint `CP2` gets executed while coming to `I5`, and the restoration logic, on the next run, restores the saved state and jumps to `CP2`. Moreover, under all possible scenarios of power failures, the output of the intermittent program remains equal to that of the continuous program.

```
Entry:
CP1: I1: push ebp
     I2: mov esp ebp
     I3: inc (nv)
CP2: I4: xor eax eax
     I5: pop ebp
     I6: ret

CP1: I1
CP2: I4
CPelems1:
  esp, (esp), nv
CPelems2:
  esp, (esp+4)
```

(a) Continuous program

```
Restoration: # new entry
      restore CPdata.CPelems CPelems
      jmp CPdata.eip  # init to Entry:
Entry:  # original entry
CP1': # checkpointing logic
      save (CPelems1, CP1) CPdata
CP1:  I1: push ebp
      I2: mov esp ebp
      I3: inc (nv)
CP2': # checkpointing logic
      save (CPelems2, CP2) CPdata
CP2:  I4: xor eax eax
      I5: pop ebp
      I6: ret
```

(b) Intermittent program

Fig. 1: The first assembly program increments a global non-volatile variable `nv` and returns `0`. It also shows the checkpoint locations `CP1` and `CP2` and respective checkpoint elements (`CPelems1` and `CPelems2`) that need to checkpointed at these locations. The second program is an intermittent program, which is generated by instrumenting the first program at the given checkpoint locations.

Notice, that we need not checkpoint the whole state of the machine, and only a small number of checkpoint elements is sufficient to ensure the equivalence with the continuous program. A smaller checkpoint is important as it directly impacts the performance of the intermittent program; a smaller checkpoint results in less time spent on saving and restoring it. Fig. 1a shows the smallest set of $CPelem$s that need to be saved at `CP1` and `CP2`. The first two elements of `CPelems1` and the only two elements of `CPelems2` ensure that the address where return-address is stored and the contents at this address, i.e., the return-address (both of which are used by the `ret` instruction to go back to the call site) are saved by the checkpoint. As per the semantics of `ret` instruction, `ret` jumps to the address stored at the address `esp`, i.e., it jumps to `(esp)`[1]. At `CP1` and `CP2`, the return address is computed as `(esp)` and `(esp+4)` respectively. Note that the expressions are different because of an intervening `push` instruction. Further, checkpointing of non-volatile data is usually not required; however, `(nv)` needs to be saved at `CP1` because it is being read and then written before the next checkpoint. If we do not save `(nv)` at `CP1`, failures immediately after `I3` would keep incrementing it.

Tools [15, 19, 26] that generate intermittent programs by automatically instrumenting the given continuous programs usually work at a higher level (C or LLVM IR). These tools perform live variable analysis for volatile state and write-after-read (WAR) analysis for non-volatile state to determine the checkpoint elements. However, they end up making conservative assumptions because of the

[1] `(addr)` represents 4 bytes of data in memory at address `addr`.

lack of knowledge of compiler transformations (e.g., unavailability of mapping between machine registers and program variables) and the proposed checkpointed elements contain unnecessary elements. For example, a tool, like DINO, without the knowledge of compiler transformations would checkpoint all the registers and all the data on the stack for our running example. Even if these analyses are ported at the machine level, the proposed checkpoint elements would be conservative as these analyses are syntactic in nature. For example, a live variable analysis for the example program would additionally propose the following unnecessary elements: `ebp` at `CP1` and `eax, (esp)` at `CP2`.

The observable of the example program is produced only at the exit. Let us consider a case, when the observable events are produced before reaching the exit (called intermediate observables). In case of intermediate observables, the observables may get produced multiple times due to the power failures. For example, assume that there is an atomic instruction `I5': print("Hello, World!")` (which produces an observable event) in between `I4` and `I5`. Due to the power failures at `I5` and `I6`, the program will again execute the code at `I5'` and the observable event will get produced again, resulting in an equivalence failure. Interestingly however, it is possible that the observer cannot distinguish, whether the observable has been repeated or not. This depends upon the semantics of `print`, e.g., if it prints to the next blank location on the console, then the observer may see multiple "Hello, World!" on the console. However, if it prints at a fixed location (line and column), then the multiple calls to `print` would just overwrite the first "Hello, World!", and this would be indistinguishable to the observer. We discuss this in detail in Sec. 5.3.

Rest of the paper is organized as: Sec. 3 presents the representation we use for abstracting programs. The modeling of intermittent program behaviour is discussed in Sec. 4. Sec. 5 describes the procedure to establish equivalence between the continuous and the intermittent program.

3 Program Representation

We represent programs as *transfer function graph*s (TFG). A TFG is a graph with nodes and edges. Nodes represent program locations and edges encode the effect of instructions and the condition under which the edges are taken. The effect of an instruction is encoded through its *transfer function.* A transfer function takes a machine state as input and returns a new machine state with the effect of the instruction on the input state. The machine state consists of bitvectors and byte-addressable arrays representing registers and memory states respectively.

A simplified TFG grammar is presented in Fig. 2. A node is named either by its program location ($pc(int)$, i.e., program counter), or by an exit location ($exit(int)$). An edge is a tuple with from-node and to-node (first two fields), its edge condition *edgecond* (third field) (represented as a function from state to expression), and its transfer function τ (fourth field). An expression ε could be a boolean, bitvector, or byte-addressable array. The expressions are similar to

$$
\begin{array}{ll}
\mathbb{T} & ::= (\mathbb{G}([node], [edge])) \\
node & ::= (pc(int) \mid exit(int), [CPelem]) \\
edge & ::= (node, node, edgecond, \tau) \\
edgecond & ::= state \rightarrow \varepsilon \\
\tau & ::= state \rightarrow state \\
state & ::= [(string, type, \varepsilon)] \\
\varepsilon & ::= const(string) \mid nry_op([\varepsilon]) \mid select(\varepsilon, \varepsilon, int) \mid store(\varepsilon, \varepsilon, int, \varepsilon) \\
CPelem & ::= (string) \mid (string, \varepsilon, int) \\
type & ::= \texttt{Volatile} \mid \texttt{NonVolatile}
\end{array}
$$

Fig. 2: Grammar of transfer function graph ($\mathbb{T}$).

the standard SMT expressions, with a few modifications for better analysis and optimization (e.g., unlike SMT, `select` and `store` operators have an additional third integer argument representing the number of bytes being read/written). An edge's transfer function represents the effect of taking that edge on the machine state, as a function of the state at the from-node. A state is represented as a set of (string, type, ε) tuples, where the string names the state-element (e.g., register name) and the type represents whether the state-element is volatile or non-volatile.

For intermittent execution, checkpoints can be inserted at arbitrary program locations. A checkpoint saves the required state elements to a persistent store. The saved state would allow the restoration logic to resume from the last executed checkpoint. We model checkpoints by annotating the TFG nodes corresponding to the checkpoint locations as *checkpoint nodes* with their corresponding checkpointed state (specified as a list $[CPelem]$ of checkpoint elements). The semantics of $CPelem$s are such that on reaching a node with $CPelem$s, the projections of $CPelem$s on the state are saved. A $CPelem$ can either specify a named register (first field) or it can specify an address with the number of bytes of a named memory (second field). The first type of $CPelem$ allows to checkpoint a register or the complete memory state, whereas the second type allows flexibility to checkpoint a memory partially or in ranges.

Fig. 3 shows the TFGs of the continuous and the intermittent programs of Fig. 1. The *edgecond* of every edge is *true* and the instructions are shown next to the edges representing the mapping between the edges and the instructions. For brevity, the transfer functions of the edges are not shown in the diagram. An example transfer function, for the instruction "`push ebp`", looks like the following: $\tau_{\texttt{push ebp}}(S) = \{S' = S;\ S'.M = store(S.M, S.esp, 4, S.ebp);\ S'.esp = S.esp - 4;\ return\ S';\}$ The new state S' has its memory and `esp` modified as per the semantics of the instruction.

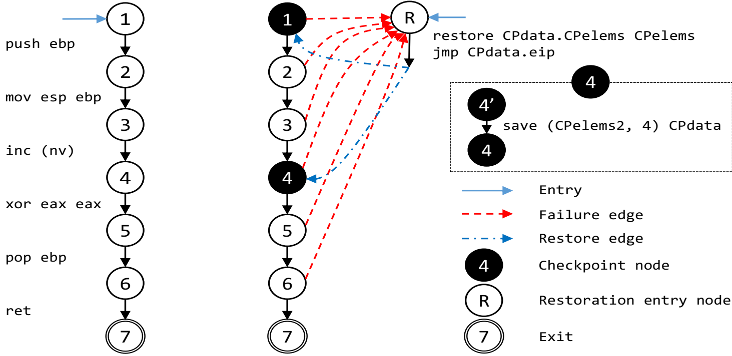


Fig. 3: TFGs of the continuous and the intermittent program of Fig. 1.

4 Modeling Intermittence

4.1 Instrumentation Model

Instrumenting a continuous program to generate an intermittent program involves: adding the *checkpointing logic* at the given checkpoint nodes, adding the *restoration logic*, changing the entry of the program to the restoration logic, and setting the initial checkpoint data in the persistent memory.

The checkpointing and the restoration logic work with data called checkpoint data (*CPdata*). The checkpoint data is read/written from/to a fixed location in a persistent memory. The checkpoint data consists of *CPelem*s of the machine state and the checkpoint location. The checkpointing logic saves the checkpoint data from the machine state, and the restoration logic updates the machine state from the checkpoint data. Additionally, after updating the machine state, the restoration logic changes the program control flow (*jmp*) to the stored checkpoint location (*CPdata.eip*). The checkpointing logic is added for all the given checkpoint nodes. The restoration logic, however, is added once, and the entry of the program is changed from the original entry to the entry of the restoration logic. The checkpoint data is initialized with the empty *CPelem* list and the stored checkpoint location is set to the original entry. This ensures that the intermittent program starts from the correct entry, i.e., the original entry, in its very first execution. Further, it is assumed that the location where *CPdata* is stored cannot alias with the addresses of the programs. In other words, the program, except for checkpointing and restoration logic, should not read or write *CPdata*.

The checkpointing logic is made atomic by using a double-buffer to save the checkpoint data. The checkpointing logic works with two checkpoint data: current *CPdata* and unused *CPdata*, and a pointer *CPdataLocation* points to the current *CPdata*. While checkpointing, it writes to the unused checkpoint data and once complete, it updates *CPdataLocation* to the address of unused checkpoint data, making it the current *CPdata*. This technique ensures that a

failure while executing the checkpointing logic does not corrupt the checkpoint data. For brevity, we do not show the implementation of double buffering.

Fig. 3 shows the TFGs of the continuous and the intermittent program. Nodes 1 and 7 are the entry and the exit locations of the continuous program respectively. In the intermittent program, the checkpointing logic has been inserted at nodes 1 and 4, and the restoration logic has been appropriately added at program entry. The *CPelem*s at node 1 (`CPelems1`) and 4 (`CPelems2`) are listed in Fig. 1a. A checkpoint node in the intermittent program is shown as a single node in the program graphs; actually, it consists of multiple nodes and edges representing the TFG of the checkpointing logic. Fig. 3 also shows the TFG of the checkpointing logic of node 4. It saves the `CPelems2` and sets the stored program location (*CPdata.eip*) to the location of the checkpoint node 4 in this example. The intermittent program always starts in the restoration logic. It restores the state from the saved *CPdata.CPelem*s and then jumps to the stored program location (*CPdata.eip*).

4.2 Modeling Power Failures

Power failures in an intermittent environment are spontaneous and can occur at any moment. We assume that a power failure can occur before and after every instruction of the assembly program, which is analogous to the properties of *precise-exceptions*, and is guaranteed by most architectures. On architectures where this assumption cannot be made, one can model power failures at the microinstruction level, i.e., before and after every microinstruction of that architecture, and rest of the technique would remain the same.

At the TFG level, nodes precisely represent the instruction boundaries, i.e., a power failure can occur at *any* of the nodes of the TFG. On a power failure: the volatile data is lost and the program, on reboot, starts from the entry, i.e, the restoration logic. We model power failures at each node by adding a non-deterministic *failure edge* from each node of the TFG to the entry of the restoration logic.

Definition 1 (Failure edge). *A failure edge is an edge of a TFG from node n to the entry node R of the restoration logic. The edgecond and the transfer function τ of a failure edge are defined as:*

$$edgecond = \delta$$

$$\tau(S) = \forall_{(s,t,\varepsilon)\in S} \begin{cases} (s,t,random_{\varepsilon}) & \textit{if } t \textit{ is } \texttt{Volatile} \\ (s,t,\varepsilon) & \textit{if } t \textit{ is } \texttt{NonVolatile} \end{cases}$$

Where δ is a random boolean value, S is the state at the node n, (s, t, ε) represents an element of the state S, and $random_{\varepsilon}$ is a random value of the type of the expression ε.

A failure edge of a TFG models the non-determinism and the effect of a power failure; the condition under which the edge is taken is random, i.e., spontaneous power failure, and the effect is modeled by the transfer function and the program

control flow change. The transfer function of a failure edge preserves the non-volatile data and garbles the volatile data (overwritten with arbitrary/random values) and the failure edge goes to the entry, encoding the fact the program starts from the entry on reboot.

The failure edges are added for all the nodes of the instrumented TFG, even for the nodes of the checkpointing and the restoration logic. The failure edges for the nodes of checkpointing and restoration logic capture the fact that power failures are possible even while executing the checkpointing and the restoration logic. This failure model is exhaustive and complete, and it precisely models the semantics of power failures. The failure edges are shown as the dashed edges in Fig. 3.

4.3 Resolving Indirect Branches of Restoration Logic

The restoration logic changes the control flow of the program based on the contents of stored program location. It is implemented as an indirect jump (i.e., `jmp CPdata.eip`) at the assembly level. In general, an indirect jump may point to any program location; however, in our case we can statically determine the set of locations the indirect jump can point to. Since the indirect jump depends on the value stored in `CPdata.eip`, we determine all the values that may get stored in `CPdata.eip`.

At the beginning, `CPdata.eip` is initialized to the original entry of the intermittent program. And later, it is only modified by the checkpointing logic and set to the locations of the checkpoint nodes. Thus, the indirect jump can either point to the original entry or any of the checkpoint nodes. Using this information, we resolve the indirect jump of the restoration logic and add *restore edges* to the intermittent TFG to reflect the same.

Definition 2 (Restore edge). *A restore edge is an edge of a TFG from the node R, i.e., the restoration logic, to the original entry or a checkpoint node n of the TFG. The edgecond and the transfer function τ of the restore edge are defined as:*

$$edgecond = (CPdata.eip == n)$$

$$\tau(S) = \forall_{(s,t,\varepsilon)\in S} \begin{cases} (s,t,\varepsilon) & (s) \notin CPdata.CPelems \\ (s,t,\varepsilon) & (s,_,_) \notin CPdata.CPelems \\ (s,t,D) & ((s):D) \in CPdata.CPelems \\ (s,t,store(\varepsilon,a,b,D)) & ((s,a,b):D) \in CPdata.CPelems \end{cases}$$

Where S is the state at the node R, (s,t,ε) is an element of the state S, (s) and (s,a,b) are checkpoint elements, $CPdata.CPelems$ has the stored checkpoint elements as a map from $CPelems$ to the stored data (D), and s, t, ε, a and b correspond to name, type, expression, address and size (number of bytes) respectively.

The edge condition represents that the edge is taken to a checkpoint node n if the stored program location $CPdata.eip$ is equal to n. The transfer function

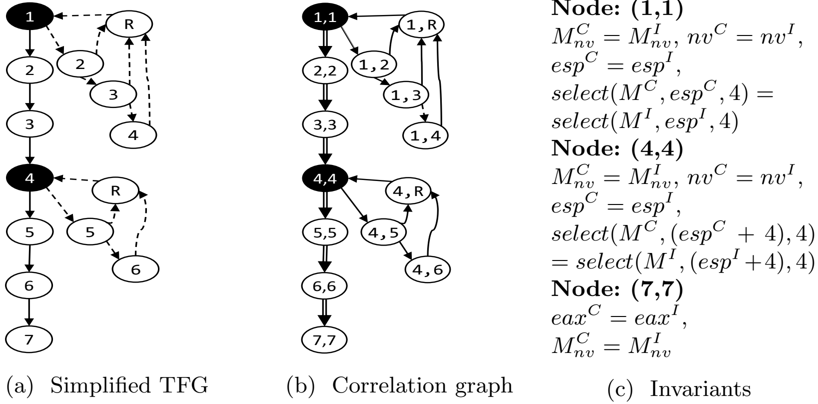
restores the state by updating the state with all the *CPelems* available in the *CPdata.CPelems*. The restore edges are added to the intermittent TFG from the restoration logic to all the checkpoint nodes and the original entry. The restore edges are shown as the dash-dot edges in Fig. 3.

5 Equivalence

Our goal is to establish equivalence between the continuous and the intermittent program, which translates to checking equivalence between the TFGs corresponding to the respective programs. Significant prior work exists for sound equivalence checking of programs in the space of translation validation and verification [4, 7, 9–14, 16, 18, 21, 23–25]. Most of these techniques try to infer a *bisimulation relation* (also called simulation relation in some papers) between the two programs. A bisimulation relation between two programs consists of *correlation* and *invariants*. The correlation is a mapping between the nodes and edges (or moves) of the two programs; the correlation sets the rules, which the two programs follow to move together in a lock-step fashion. The invariants relate the variables across the two programs, at the correlated node pairs. The invariants always hold when the two programs are at the respective correlated nodes. Further, the invariants should prove the above-mentioned correlation and equivalence of the observables of the two programs on the correlated edge pairs.

Prior work on equivalence checking has proposed different algorithms to infer the correlation and invariants, that work in different settings and with different goals. Because our equivalence problem is unique, we cannot just offload it to any existing equivalence checker. The important differences that make this problem unique are: (1) The intermittent program, which runs in an environment with power failures, has non-determinism whereas the continuous program is deterministic. Previous techniques work in a setting where both the programs are deterministic, unlike ours, where one of the programs (the intermittent program) has edges that can be taken non-deterministically, i.e., the failure edges. Consequently, the correlation is different as power failures would be now modeled as *internal* moves, and hence we instead need to infer a *weak bisimulation relation* [20]. (2) Due to recurring executions in the intermittent program (because of the power failures), an intermediate observable event in the intermittent program can be produced more times than in the continuous program. To reason about the same, we describe two properties of the observables, namely *idempotence* and *commutativity* and we use them to establish equivalence under repeated occurrences of the observables.

As we have seen in Fig. 1, the amount of instrumentation code added to intermittent program is quite small and most of the code of the intermittent program remains the same. However, even in this setting, the problem of checking equivalence between the continuous and the intermittent program is undecidable in general. In other words, determining whether a certain checkpoint element (*CPelem*) needs to be checkpointed is undecidable. We define equivalence between a continuous and an intermittent program, i.e., across the in-



(a) Simplified TFG (b) Correlation graph (c) Invariants

Fig. 4: The first figure shows a simplified intermittent TFG, the edges and the nodes have been duplicated for exposition and non-reachable failure paths have been removed. Checkpoint-to-checkpoint paths formed by dashed edges are failure paths and that formed by solid edges are progress paths. The second figure shows the correlation graph; single-edges show correlations of no-moves with failure paths. The third figure shows the invariants at the checkpoint nodes and exit.

strumentation, and we prove the theorem that determining this equivalence is undecidable.

Definition 3 (Equivalence). *A continuous TFG (C) is equivalent to an intermittent TFG (I), where I has been generated by instrumenting C, if starting from identical input state S, the two TFGs produce equivalent observable behaviour, for all values of S.*

Theorem 1. *Given a continuous TFG (C) and an intermittent TFG (I), where I has been generated by instrumenting C, determining equivalence between C and I is undecidable.*

Proof. Determining whether any function `f` halts can be reduced to this problem. Consider the following construction of a continuous (C) and an intermittent (I) program: `C(a)={f(); print(a);}` `I(a)={CP(); f(); print(a);}`, such that `CP()` checkpoints the complete state except the volatile variable `a`. The two functions can only be equivalent if `f()` does not halt. Checking whether `f` halts can be written in terms of determining whether the two functions are equivalent: `f_Halts` = (`C` $\neq$ `I`). However, the halting problem is undecidable, hence, checking equivalence between a continuous and an intermittent program is also undecidable.

5.1 Correlation

The correlation across two TFGs defines a mapping between the nodes and the paths (also called moves) of the two TFGs. It tells the path taken by one program,

if the other program takes a certain path, and vice versa. In our case, we reason in terms of the paths from one checkpoint to another checkpoint (*checkpoint-to-checkpoint paths*, defined next) and define the correlation in terms of the same.

Definition 4 (Checkpoint-to-checkpoint path). *Given a continuous TFG C and an intermittent TFG I, where I has been generated by instrumenting C: a path from node n to node m in the intermittent TFG I is a checkpoint-to-checkpoint path if the nodes n and m belong to the set $N = \{entry, exit\} \cup CPs$, and none of its intervening nodes between n and m belongs to N. Here entry, exit and CPs are the original entry, the exit and the set of checkpoint nodes respectively.*

A checkpoint-to-checkpoint path in the continuous program C is defined in the same manner, however, assuming the checkpoint nodes of the corresponding intermittent TFG (i.e., I); this is because C has no notion of checkpoint nodes.

The checkpoint-to-checkpoint paths are further classified depending upon whether a power failure occurs or not, on a checkpoint-to-checkpoint path.

Definition 5 (Failure path). *A checkpoint-to-checkpoint path is a failure path if a power failure occurs in it.*

Theorem 2. *A failure path starts and terminates on the same checkpoint. In other words, a failure path starting and terminating on different checkpoint is not reachable.*

Proof. Since there are no intervening checkpoints on a failure path, the stored checkpoint location (*CPdata.eip*) is the starting checkpoint (n), implying that on a failure, only one restore edge, which goes from the restoration logic to the starting checkpoint, will have its *edgecond true*.

Definition 6 (Progress path). *A checkpoint-to-checkpoint path is a progress path if there are no power failures in it.*

A checkpoint-to-checkpoint path starting from a checkpoint can either reach a successive checkpoint if no power failure occurs in between, or it reaches back to the starting checkpoint (via a failure and then the restore edge to it) if there is a power failure. A checkpoint-to-checkpoint path in the intermittent TFG is either a failure path or a progress path. However, all the checkpoint-to-checkpoint paths in the continuous program are progress paths as there are no failures in it. Fig. 4a shows the failure and the progress paths of the intermittent TFG. Note that we have not shown the edges of the TFG of checkpointing logic, we get rid of them by composing these edges with the incoming edges of the start node of a checkpoint, e.g., path $3 \rightarrow 4' \rightarrow 4$ is collapsed into an edge $3 \rightarrow 4$.

We use the notion of a *weak bisimulation relation* [20] to establish equivalence between the continuous and the intermittent TFGs. The non-deterministic failure paths of the intermittent TFG are modeled as the *internal* moves and progress paths of the two TFGs are treated as the usual moves. We propose the following correlation between the two TFGs:

Definition 7 (Correlation). *Given a continuous TFG C and an intermittent TFG I, where I has been generated by instrumenting C, both starting from the original entry or the same checkpoint node (n_{CP}):*

1. *If I takes a progress path p, then C takes the corresponding progress path p in it, and vice versa. Additionally, the individual edges of the progress paths are also taken together. That is, if C takes the edge $(n \to m) \in p$, then I takes the same edge $(n \to m)$, and vice versa. That is, for all nodes $n \in p$ and edges $(n \to m) \in p$: node n and edge $n \to m$ of C are correlated with node n and edge $n \to m$ of I, respectively.*
2. *If I takes a failure path p, then C takes a* no-move*, i.e., C does not move at all and stays at the same node (n_{CP}), and vice versa. Further, every individual edge of the failure path of I is taken with a no-move of C. That is, for all nodes $n \in p$: node n of I is correlated with node n_{CP} of C.*

Intuitively, the above correlation of moves states that for TFGs starting from the entry or the same checkpoint: if there is no power failure, and the intermittent program moves to a successive checkpoint, then the continuous program also moves to the same next checkpoint, and vice versa. However, if the intermittent program undergoes a failure, hence, returning to the starting checkpoint, then the continuous program does not move at all, and stays at the starting checkpoint, and vice versa.

Correlation between two TFGs forms a graph, whose nodes and edges are pairs of nodes and edges of the two TFGs. That is, if (n_C, n_I) is a node and (e_C, e_I) is an edge of the correlation graph, then n_C and n_I are the nodes of the continuous and the intermittent TFG respectively, similarly, e_C and e_I are the edges of the continuous and the intermittent TFG respectively. Fig. 4b shows the correlation graph for the running example.

5.2 Inferring Invariants

Once we have fixed the correlation between the two TFGs, we need to check if the correlation is indeed correct as it is not necessary that the two TFGs take the same progress path starting from the same checkpoint. Furthermore, we need to check that the two TFGs produce the same observable behaviour. This involves inferring invariants at the nodes of the correlation graph. These invariants are also called coupling predicates [7] as they relate the variables of the two programs. The invariants at some node (n_C, n_I) of the correlation graph should always hold if the continuous TFG is at node n_C and the intermittent TFG is at node n_I.

The inferred invariants should be strong enough to prove that the correlated edges are taken together and the observables at the correlated edges are identical. Formally:

$$\forall_{(n_C,n_I)\to(m_C,m_I)} \; invariants_{(n_C,n_I)} \Rightarrow_{(n_C,n_I)\to(m_C,m_I)} (o_{(n_C\to m_C)} = o_{(n_I\to m_I)}) \wedge$$
$$(edgecond_{(n_C\to m_C)} = edgecond_{(n_I\to m_I)})$$

Here $(n_C, n_I) \to (m_C, m_I)$ is an edge in the correlation graph, $n_C \to m_C$ and $n_I \to m_I$ are edges in the continuous and the intermittent TFG respectively, $invariants_{(n_C,n_I)}$ represents the conjunction of the invariants at the correlation node (n_C, n_I), $edgecond_e$ represents the edge condition of an edge e, o_e represents the observable on an edge e, and $\Rightarrow_{(n_C,n_I)\to(m_C,m_I)}$ represents the implication over the edge $(n_C, n_I) \to (m_C, m_I)$.

We employ a Houdini like [8] guess-and-check technique to infer invariants of the correlation graph. Candidate invariants are generated on each node of the correlation graph, based on certain rules, and a checking procedure then eliminates the invalid candidate invariants. At the end, we are left with valid invariants and use them to verify the correctness of the correlation graph and the equivalence of observables.

We generate candidate invariants based on the following simple rule: For every node (n_C, n_I) of the correlation graph, all possible predicates of the form $s_C = s_I$ are generated for every s_C and s_I that are read or written in the corresponding TFGs, where s_C and s_I are state elements in states S_{n_C} and S_{n_I} respectively. Intuitively, the partial states (only the state that is read or written) of the two programs are equated at the correlated nodes.

The checking procedure is a fixed-point computation that keeps eliminating the incorrect invariants until all the incorrect invariants are eliminated. On every edge of the correlation graph, the checking procedure tries to prove an invariant at the to-node, using the invariants at the from-node, and if the invariant is not provable, then it is eliminated. The procedure terminates if no more invariants can be eliminated. Further, the invariants at the entry node are proven using the condition that the states of the two TFGs are equivalent at entry, i.e., equal inputs. Formally, we apply the following check:

$$(S_{entry_C} = S_{entry_I}) \Leftrightarrow invariants_{(entry_C, entry_I)}$$

$$\underset{(n_C,n_I)\to(m_C,m_I)}{\forall} invariants_{(n_C,n_I)} \Rightarrow_{(n_C,n_I)\to(m_C,m_I)} invariant_{(m_C,m_I)}$$

Here $invariant_{(m_C,m_I)}$ is a candidate invariant at node (m_C, m_I), S_{entry_C} is the state at the entry node of the TFG C, and $invariants_{(n_C,n_I)}$ is the conjunction of the current set of (not eliminated) candidate invariants at node (n_C, n_I).

Fig. 4c shows the inferred invariants for some nodes, which can prove the required conditions and equivalence for the running example, under the $CPelems$ of Fig. 1a.

On final notes, our equivalence checking algorithm is general and handles loops seamlessly; in fact, we are already handling the loops which get introduced due to the failure edges. Had there been a loop in the example program, say there is a backedge from node 3 to 2, it would reflect in the failure and progress paths too, e.g., Fig. 4a will contain a solid as well as a dash edge from node 3 to 2. Similarly, the correlation graph too will have edges from node (3,3) to (2,2) and node (1,3) to (1,2). Also, all the benchmarks that we used for evaluation contain one or more loops. Finally, our technique is not without limitations, it is possible that a correlation other than the proposed one, or an invariant of

different shape/template (other than the one used) is required for proving the equivalence. Though we did not encounter this in practice.

5.3 Intermediate Observables

We now discuss the issue with observables occurring at the intermediate nodes, i.e., the nodes other than the exit node. We call these the intermediate observables. In an intermittent program, an intermediate observable event can be produced more times than is produced in the continuous program. It happens because of the recurring executions of an intermediate observable due to power failures. Given a sequence $\lambda^C = o_1 o_2 ... o_i ... o_x$ (written as a string) of observable events on a progress path (from checkpoint node n_1 to checkpoint node n_{x+1}) of the continuous TFG, the event o_i is produced on the edge $n_i \rightarrow n_{i+1}$, for $i \in [1, x+1)$. The possible sequences of observable events for the corresponding intermittent TFG, during the moves from checkpoint node n_1 to checkpoint node n_{x+1} (by taking one or more failure paths followed by a progress path) are:

$$\lambda^I = \lambda^I_{n_1} \lambda^C \quad \textit{such that} \;\; \lambda^I_{n_1} = (o_1 | o_1 o_2 | o_1 o_2 o_3 | ... | o_1 o_2 ... o_{x-1})^*$$

The sequence is written as a regular expression, where $*$ represents Kleene start, i.e., zero or more repetitions and | represents alternation. The first part of the expression $\lambda^I_{n_1}$ represents all sequences of observables produced at node n_1. The alternation operator encodes that a failure may happen at any node n_i and may produce a sequence $o_1 o_2 ... o_{i-1}$ for $i \in [1, x]$ (a failure at n_{x+1} will not take back to n_1); the $*$ operator encodes that failures may occur zero or more times. The second part (λ^C) represents the case when there is no power failure and the execution reaches the successive checkpoint n_{x+1}.

The sequence of observables produced in the intermittent program could be different from that produced in the continuous TFG. However, if the effects of the two sequences, i.e., λ^C and λ^I are same, and the observer cannot differentiate between the two, we will be able to claim the equivalence of the two programs. To this end, we define a notion of *idempotence* and *commutativity* of observables, and we use these properties to prove that the sequences of observables produced by the continuous and the intermittent TFG are equivalent if the observables are idempotent and commutative.

Definition 8 (Idempotence). *On observable event o is idempotent if its recurring occurrences are undetectable to the observer. That is, the sequence oo produces the same effect as o.*

Definition 9 (Commutativity). *The observable events o_1 and o_2 are commutative if the order of occurrences of the two events is not important to the observer. That is, the sequences $o_1 o_2$ and $o_2 o_1$ are both equivalent to the observer.*

Intuitively, an observable is idempotent if the observer cannot detect if the observable occurred once or multiple times. For example, the observable `print(line,`

`column, text)`, which prints `text` at the given `line` and `column`, is idempotent. The user cannot distinguish if multiple calls to this function have been made. Observables `setpin(pin, voltage)` (sets the voltage of the given pin) and `sendpkt()` (send network packet) are more examples of idempotent observables. The observer cannot tell if the function `setpin()` is called multiple times, as it will not change the voltage of the pin on repeated executions. In case of `sendpkt()`, if the network communication is designed to tolerate the loss of packets, and consequently, the observer/receiver is programmed to discard the duplicate packets, then the observable is idempotent with respect to the receiver. Two observables are commutative if it does not matter to the observer, which one occurred first. For example, if a program lights an LED and sends a packet, and if these two events are independent to the observer, e.g., the packet is meant for some other process and the LED notification is meant for the user, then their order is unimportant to the observer.

Theorem 3. *$\lambda^I = \lambda^C$, if for all o_i and o_j in λ^C, o_i is idempotent, and o_i and o_j are commutative.*

Proof. In sequence λ^I, we move an event o_i to position i (by applying commutativity) and if the same event is present at $i + 1$, we remove it (by applying idempotence), we keep applying these steps until only one o_i remains. Performing these steps in increasing order of i, will transform λ^I into λ^C. If the length of λ^I is finite, termination is guaranteed.

With all the pieces, we state the final theorem now:

Theorem 4. *A continuous TFG C and an intermittent TFG I, where I is generated by instrumenting C, are equivalent if:*

1. *Invariants can prove the correlation and the equivalence of observables at each correlated edge of the progress paths (Sec. 5.2).*
2. *On every progress path: each observable is idempotent, and every pair of observables is commutative (Sec. 5.3).*
3. *Both the TFGs, i.e., C and I, terminate.*

Proof. Proof by induction on the structure of programs:
Hypothesis: Both programs C and I produce the same observable behaviour on execution till a node n, for $n \in N = \{entry, exit\} \cup CPs$, where CPs is the set of checkpoint nodes.
Base: At entry, the two C and I have same observable behaviour.
Induction: Assuming the hypothesis at all the immediate predecessor checkpoints (m) of node (n), we prove that the observable behaviour of the two programs are equivalent at n, where $m, n \in N$.
Observable sequence at node n for program I can be written in terms of the observable sequence at the predecessor node m and the observable sequence produced during the moves from m to n: $\lambda^I_n = \lambda^I_m \lambda^I_{m \to n}$. From Condition#1, we can prove that the two programs move together from m to n and the individual observables of the two programs are same. Using the same along with Condition#2, Condition#3 and Theorem 3, we claim that $\lambda^I_{m \to n} = \lambda^C_{m \to n}$. Finally, using the hypothesis $\lambda^I_m = \lambda^C_m$, we prove that $\lambda^I_n = \lambda^C_n$.

6 Evaluation

We evaluate our technique in terms of the runtime of verification, and the robustness and capability of our algorithm. We are not aware of any previous verifier for this problem, and so we do not have a comparison point for the verification runtimes of our tool. However, we do compare the robustness and capability of our technique by using our verifier in a simple synthesis loop, whose goal is to minimize the size of checkpoints at a given set of checkpoint nodes. Moreover, the capability of this synthesis loop is dependent on the capability of our verifier. If our verifier can prove the equivalence between the continuous and the intermittent programs, with smaller checkpoints, then the synthesis loop can generate an intermittent program with smaller checkpoints. This also enables us to compare our work with DINO [15]. With similar goals, DINO automatically generates an intermittent program from a given continuous program by instrumenting it at a given set of checkpoint locations. It works with mixed-volatility programs and performs a syntactic analysis to determine the checkpoint elements that need to be checkpointed. However, unlike our tool, DINO's output is unverified. A detailed comparison with DINO is available in Sec. 7.

We implemented our equivalence checking technique in a verifier for the x86 architecture. Our technique is independent of the architecture, the reason why the x86 architecture was chosen is that we had access to a disassembler and semantic modeling of x86 ISA. Constructing a TFG from an executable required us to resolve other indirect jumps (other than that of the restoration logic) occurring in the program, in particular, the indirect jumps due to the function returns, i.e., the `ret` instructions. A `ret` instruction takes back the program control to the return-address stored in a designated location in the stack. The return-address is set by the caller using the `call` instruction. We perform a static analysis to determine the call sites of every function and hence determine the return-addresses of every `ret` instruction. We appropriately add the *return edges* (similar to restore edge) from the return instruction to the determined call sites. The transfer function of the return edge is *identity* and its $edgecond = (return_address == call_site_address)$.

While testing our verifier on some handwritten pairs of continuous and intermittent programs, we found that it is very easy for a human to make mistakes in suggesting the checkpoint elements and checkpoint locations, especially for mixed-volatility programs. For example, in the example program, the user ought to specify a checkpoint before `I3`. If a checkpoint location is not specified before `I3`, the intermittent program cannot be made equivalent to the continuous program no matter what the checkpoint elements are. Our verifier gets used by the synthesis loop, and the average runtime of our verification procedure ranges between 1s to 332s for benchmarks taken from previous work on intermittent computation [15, 19]. Tab. 1 describes our benchmarks and results, and the seventh column shows the individual average runtimes for different benchmarks. Almost all the verification time is spent on checking satisfiability of SMT queries. We discharge our satisfiability queries through the Yices SMT solver [6].

Benchmark	# CP nodes	Avg. CP size DINO	Avg. CP size synthesis loop	Improvement over DINO	Synthesis time (s)	Avg. verification runtime
DS	5	120.8	42.4	2.8x	3500	16.5
MIDI	4	80	19	4.2x	2154	11.9
AR	2	128	22	5.8x	26290	332.8
CRC	2	96	24	4x	42	1.1
Sense	3	96	25.3	3.8x	331	3.2

Table 1: For each benchmark, the second column gives the number of checkpoint nodes, the third and the fourth column give the average checkpoint size (bytes) determined by DINO and synthesis loop respectively, the fifth column gives improvement by synthesis loop over DINO, and the sixth and the last column give the total time taken by the synthesis loop and the average runtime of the verifier respectively.

We implemented a synthesis loop to optimize the checkpoint size. Given a set of checkpoint locations, the synthesis loop tries to greedily minimize the checkpoint elements that need to be checkpointed. It keeps proposing smaller checkpoints (with fewer $CPelems$), and it relies on our verifier to know the equivalence between the continuous and the intermittent program, with the current checkpoint elements. The synthesis loop starts by initializing each checkpoint node with all possible checkpoint elements (the most conservative solution). It then iterates over each checkpoint element of all the checkpoint nodes, and considers each checkpoint element for elimination. It greedily removes the current checkpoint element if the remaining checkpoint elements preserve equivalence. The loop terminates after considering all the checkpoint elements and returns the last solution. Clearly, the capability of this synthesis loop is dependent on the robustness and capability of the verifier. If the verifier can verify intermittent programs with fewer checkpoint elements, only then can the synthesis loop can result in a better solution.

We took benchmarks from previous work [15, 19] (all the DINO benchmarks are included) and used the synthesis loop and DINO to generate checkpoint elements at a given set of checkpoint nodes. For each benchmark, Tab. 1 shows the size of checkpoints generated by the synthesis loop and DINO for the same set of checkpoint nodes. The synthesis loop is able to generate checkpoints with 4x improvement over DINO, i.e., the data (in bytes) that needs to be checkpointed is on average 4 times less than that determined by DINO. The synthesis loop is able to perform better than DINO because of the precision in the model of the intermittent programs and the precision that we get while working at the assembly level (Sec. 7). Additionally, the synthesis loop benefits from the semantic reasoning over the syntactic reasoning done by DINO (Sec. 2).

7 Related Work

We compare our work with the previous work on automatic instrumentation tools that generate intermittent programs, namely DINO [15], Ratchet [26] and

Mementos [19]. These tools work in different settings and employ different strategies for checkpointing. In contrast, our work is complementary to these tools, and our verifier can be employed to validate their output.

DINO works with mixed-volatility programs, and given the checkpoint locations, it generates the intermittent programs automatically. It proposed a control flow based model of intermittence, where the control flow is extended with failure edges, going from all the nodes to the last executed checkpoints. This modeling is conservative and incomplete as it lacks semantics and does not model the effect of the power failures, unlike ours, where the failure edge is defined formally, in terms of the edge condition and the transfer function of a failure edge. Consequently, the model is not suitable for an application like equivalence checking. It then performs a syntactic WAR analysis (write-after-read without an intervening checkpoint) of non-volatile data on this extended control flow graph to determine the non-volatile data that needs to be checkpointed. Since it works at a higher level and does not have a mapping between the machine registers and the program variables, it ends up checkpointing all the registers and all the stack slots resulting in unnecessary checkpoint elements. Further, DINO does not work with intermediate observables and the output is not verified. Our work is complementary to DINO, in that our verifier can be used to validate DINO's output.

Ratchet is a fully-automatic instrumentation tool to generate intermittent programs from continuous programs. However, it takes a radically different approach of assuming that the whole memory is non-volatile, i.e., all program data including the stack and heap are deemed non-volatile. Only the machine registers are assumed to be volatile. Ratchet works by adding a checkpoint between every WAR occurrence on non-volatile data, i.e., it breaks every WAR occurrence. By breaking every WAR occurrence, correctness of non-volatile data across power reboots is ensured; for the machine registers, Ratchet simply saves the live machine registers at every checkpoint. These simplifications involve a performance cost, as it results in frequent checkpoints because the checkpoint locations are now determined by these WAR occurrences. Further, it is not possible to insert a checkpoint between WAR occurrences within a single instruction (e.g., "`inc (nv)`"). Ratchet authors also do not allow intermediate observables. Finally, Ratchet's output can also be verified using our tool.

Mementos is a hardware-assisted fully-automatic instrumentation tool to generate intermittent programs. At each checkpoint location, it relies on hardware to determine the available energy and the checkpointing logic is only executed if the available energy is less than a threshold level, i.e., the checkpoints are conditional. Interestingly, our verifier does not require any modification to work in this setting, the only difference would be that the number of failure and progress paths that get generated would be more. A checkpoint-to-checkpoint path can now bypass a successive checkpoint, resulting in a checkpoint-to-checkpoint path to a second level successor. For example, in our example, there will be also a progress path from node 1 to the exit, because the checkpoint at node 4 is conditional.

Systems that tolerate power failures are not uncommon, file system is one example that is designed to tolerate power failures. The file system design has to ensure that across power failures, the disk layout remains consistent. In addition to power failures, it has to worry about disk write reorderings done by the disk controller. FSCQ [2] and Yggdrasil [22] are two recent papers that formally verified the file systems under power failures and reorderings. FSCQ is written in Coq and requires manual annotations and proofs for verification. Yggdrasil, on the other hand is an automatic technique. In FSCQ, the specifications are given in Crash Hoare Logic (CHL) which allows programmers to specify the expected behaviour under failures. The verification then entails proving that the file system follows the given specifications. In Yggdrasil, the behavioral specifications are provided as higher-level programs; The verification involves checking whether the file system is a *crash refinement* of the given specification, i.e., it produces states that are a subset of the states produced by the specification. The specifications in both the techniques are crash-aware, i.e., the specification encodes the behaviour under power failures. In contrast, our specifications are continuous programs and are not aware of crashes, the intermittent should behave as if there are no power failures. In addition, the problem of intermediate observables is unique to our setting. It would be interesting to explore if our technique can be used to verify file systems. Considering that our technique works smoothly with loops, it would remove Yggdrasil's important shortcoming of its inability to reason about loops in a uniform way.

Smart card embedded systems are another interesting example of systems that are designed to work with failures. These cards get powered by inserting in the appropriate terminal, and suddenly removing it during an operation may leave the card's data in an inconsistent state. A mechanism is added to restore a consistent state on the next insertion. A card has anti-tearing properties if it can always be restored to a consistent state after tearing (removal) at every execution state. Anti-tearing properties of smart cards are important and previous work [1] formally verifies this by proving that tearing is safe at every program point in Coq. This technique is not automatic and requires manual proofs.

Our work overlaps with previous work on equivalence checking in the context of translation validation and verification [4,5,7,9–14,16,18,21,23–25]. The goal of translation validation is to compute equivalence across compiler optimizations. On the other hand, our work targets equivalence across the instrumentation, albeit, under power failures. We have borrowed ideas from previous work, e.g., invariant inference is similar to that of [3–5] which are further based on Houdini [8]. However, tackling non-determinism due to power failures and the problem with intermediate observables is perhaps new to this space.

To conclude, we present a formal model of intermittence and a technique to verify the correctness of the intermittent programs with respect to their continuous versions. Our experiments demonstrate that synthesis along with working at the binary level can reduce the size of the checkpoints significantly. We hope that automatic instrumentation tools can leverage these ideas to produce verified and efficient intermittent programs.

References

1. Andronick, J.: Formally proved anti-tearing properties of embedded c code. Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006) pp. 129–136 (2006)
2. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. SOSP '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2815400.2815402
3. Churchill, B., Sharma, R., Bastien, J., Aiken, A.: Sound loop superoptimization for google native client. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 313–326. ASPLOS '17, ACM (2017)
4. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Proceedings of the Fifteenth Asian Symposium on Programming Languages and Systems. pp. 127–147. APLAS '17 (2017)
5. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017. pp. 19–34. HVC '17 (2017)
6. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV'2014). Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (July 2014)
7. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 349–360. ASE '14, ACM, New York, NY, USA (2014)
8. Flanagan, C., et al.: Houdini, an annotation assistant for esc/java. In: FME 2001: Formal Methods for Increasing Software Productivity, LNCS, vol. 2021, pp. 500–517. Springer Berlin Heidelberg (2001)
9. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 327–337. PLDI '09, ACM, New York, NY, USA (2009)
10. Lahiri, S., Hawblitzel, C., Kawaguchi, M., Rebelo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: CAV '12. Springer (July 2012)
11. Lahiri, S., Sinha, R., Hawblitzel, C.: Automatic rootcausing for program equivalence failures in binaries. In: Computer Aided Verification (CAV'15). Springer (July 2015)
12. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. PLDI '03 (2003)
13. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 364–377. POPL '05, ACM, New York, NY, USA (2005)
14. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 22–32. PLDI 2015, ACM, New York, NY, USA (2015)

15. Lucia, B., Ransford, B.: A simpler, safer programming and execution model for intermittent systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 575–585. PLDI '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2737924.2737978
16. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. pp. 83–94. PLDI '00, ACM, New York, NY, USA (2000)
17. Olivo, J., Carrara, S., Micheli, G.D.: Energy harvesting and remote powering for implantable biosensors. IEEE Sensors Journal 11(7), 1573–1586 (July 2011)
18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 151–166. TACAS '98, Springer-Verlag, London, UK, UK (1998)
19. Ransford, B., Sorber, J., Fu, K.: Mementos: System support for long-running computation on rfid-scale devices. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 159–170. ASPLOS XVI, ACM, New York, NY, USA (2011)
20. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York, NY, USA (2011)
21. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 391–406. OOPSLA '13, ACM, New York, NY, USA (2013)
22. Sigurbjarnarson, H., Bornholt, J., Torlak, E., Wang, X.: Push-button verification of file systems via crash refinement. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 1–16. OSDI'16, USENIX Association, Berkeley, CA, USA (2016), http://dl.acm.org/citation.cfm?id=3026877.3026879
23. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, Lecture Notes in Computer Science, vol. 4171, pp. 496–501. Springer Berlin Heidelberg (2008)
24. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 264–276. ACM, New York, NY, USA (2009)
25. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–305. PLDI '11, ACM, New York, NY, USA (2011)
26. Van Der Woude, J., Hicks, M.: Intermittent computation without hardware support or programmer intervention. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 17–32. OSDI'16, USENIX Association, Berkeley, CA, USA (2016)

On abstraction and compositionality for weak-memory linearisability ⋆

Brijesh Dongol[1], Radha Jagadeesan[2], James Riely[2], and Alasdair Armstrong[1,3]

[1] Brunel University of London, UK
[2] DePaul University, Chicago, USA
[3] University of Cambridge, Cambridge, UK

Abstract. Linearisability is the de facto standard correctness condition for concurrent objects. Classical linearisability assumes that the effect of a method is captured entirely by the allowed sequences of calls and returns. This assumption is inadequate in the presence of relaxed memory models, where *happens-before* relations are also of importance.
In this paper, we develop hb-linearisability for relaxed memory models by extending the classical notion with happens-before information. We consider two variants: *Real time* hb*-linearisability*, which adopts the classical view that time runs on a single global clock, and *causal* hb*-linearisability*, which eschews real-time and is appropriate for systems without a global clock. For both variants, we prove *abstraction* (so that programmers can reason about a client program using the sequential specification of an object rather than its more complex concurrent implementation) and *composition* (so that reasoning about independent objects can be conducted in isolation).

1 Introduction

An implementation is linearisable [19] if for every history of the implementation, there exists a legal history of the specification such that (1) each thread makes the same method invocations in the same order, and (2) the order of non-overlapping invocations is preserved. This notion of linearisability intuitively ensures that each method invocation takes effect between its invocation and response events. Thus, instead of complex concurrent reasoning that requires a characterisation of all possible interactions across method invocations, linearisability ensures that every method call can be understood in isolation via preconditions and postconditions, as familiar in sequential computing.

Linearisability is a local property. Thus, we can reason *compositionally* about a system; i.e., to prove the linearisability of the whole, it suffices to prove the linearisability of projections to components with disjoint memories. This ability to decompose large linearisability proofs is critical to scale the use of linearisability in concurrent reasoning.

⋆ This work was partially supported by EPSRC grants EP/N016661/1 and EP/K008528/1, and NSF Grant No. 1617175.

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 183–204, 2018.
https://doi.org/10.1007/978-3-319-73721-8_9

In this paper, we are also interested in understanding *abstraction* or *contextual refinement* for correctness conditions, i.e., what correctness conditions allow a concurrent object implementation CS to be substituted for an abstract specification AS within a client program? Specifically, we would like to find a condition Z between AS and CS for which

$$Z(AS, CS) \Rightarrow \forall C : Client.\ C[AS] \sqsubseteq C[CS] \qquad (\dagger)$$

holds, where $\sqsubseteq$ denotes some notion of *refinement*, $C[AS]$ denotes a client C that uses the abstract object AS, and $C[CS]$ denotes a client C that uses the implementation object CS.

There are different solutions to (†), depending on the notion of refinement that one uses. Filipović et al. [16] study contextual refinement for terminating computations. They establish refinement between the initial and final states of $C[AS]$ and $C[CS]$ by showing that (†) holds if Z is instantiated to *linearisability*. Others have studied contextual refinement for *traces* of $C[AS]$ and $C[CS]$, where Z must be strengthened to cope with liveness properties [15, 17, 29, 22].

The works cited above assume that threads communicate via *sequentially consistent* (sc) memory [21], where memory events appear to occur according to a single, global, total order consistent with program order. However, high-performance multicore systems typically implement *relaxed* memory models, where memory events may appear to occur out-of-order with respect to program order [1, 2, 5, 6, 23, 25–27]. Under sc, client memory events that occur before a method call in program order cannot overlap with client memory events that occur after the method call (in program order). Under relaxed memory, this property fails to hold.

The impact of relaxed memory in the specification of concurrent data structures is already seen in practice via the explicit specification of *happens-before* (hb) information, e.g., consider the ConcurrentQueue in java.util.concurrent[4]. In addition to the usual guarantee — "The ConcurrentLinkedQueue class supplies an efficient scalable thread-safe non-blocking FIFO queue" — the specification of this class also describes *memory consistency effects* — "As with other concurrent collections, actions in a thread prior to placing an object into a ConcurrentLinkedQueue *happen-before* actions subsequent to the access or removal of that element from the ConcurrentLinkedQueue in another thread." This pattern is repeated for all the classes in this package.

In this paper, we study correctness criteria for data structures in the presence of such memory effects.

- We provide a formalisation of sequential specifications that have been augmented with happens-before information. This augmentation is essential for compositional reasoning. Classical linearisability does not mention *happens-before* information (since it assumes sc). Following [20], we demonstrate by

[4] This package contains data structures and utilities for concurrent programming in Java. See `https://docs.oracle.com/javase/9/docs/api/index.html?java/util/concurrent/package-summary.html`.

example that the traditional perspective fails to ensure contextual refinement for threads that communicate via relaxed memory.
- We define *real-time* hb-*linearisability*, strengthening linearisability to preserve happens-before.
- Concurrent programming in Java-like languages eschews a notion of *global* time. Rather, the idea is that a programmer specifies ordering constraints by defining how threads communicate with explicit mechanisms, such as locks, to ensure that actions taken by one thread would be seen in a reasonable way by other threads [23]. We define *causal* hb-*linearisability* to more directly model this partial order perspective on executions.

Real-time hb-linearisability is stronger than classical linearisability. As we shall demonstrate, causal hb-linearisability and real-time hb-linearisability are incomparable.

Both notions of hb-linearisability are defined relative to a memory model. For both, we show that contextual refinement holds for any relaxed memory model that satisfies the axioms[5] of Alglave, Maranget and Tautschnig (AMT) [3], which are summarised in Section 2. One of our key contributions is the enhancement of AMT to account for events arising from method invocations and responses (Sections 4). We show that, under mild assumptions, any linearisable implementation of concurrent collection must already satisfy the extra happens-before required by a specification; thus discharging the additional proof obligations incurred when proving correctness relative to real-time hb-linearisability.

We provide motivational examples in Section 3. The main definitions and results follow in Sections 4 and 5, respectively.

2 Background: AMT axioms

AMT provide an exhaustive study of relaxed memory models in [3]. Impressively, they manage to capture the details of several specific architectures (including TSO, ARMv7 and Power) in a general framework. They provide a list of axioms that are satisfied by all of the architectures they consider. Fortunately, these axioms are sufficient to establish our results. In this section, we describe the core components of this framework; we refer the interested reader to the original paper [3] for further details.

Let $\mathbb{E}$ be a set of *events*. Each event e is a tuple consisting of a unique identifier, $\mathsf{id}(e)$, a thread identifier, $\mathsf{thread}(e)$, an *action*, and other data. Actions include *memory actions*, e.g., reads and writes; other actions are architecture dependent, including fences. The axioms take as input six relations over $\mathbb{E}$, which together define an *execution*.

- po (program order), which defines a total order on the events of each thread. Events of different threads are unrelated.

[5] While we state our results relative to the axioms of Alglave et al., the ideas behind real-time and causal hb-linearisability can be applied to any other axiomatic memory model based on partial orders.

- co (coherence order), which is a total order on the writes of each location.
- rf (reads from), which maps writes to reads. Each read must be associated with exactly one write, but a write may map to more than one read.
- ppo (preserved program order), which is a suborder derived from po by removing order between events that commute according to the architecture.
- fences, which relates events in po that are separated by a fence.
- prop (propagation order), which relates writes that must propagate to memory in a particular order.

Program order only relates events of the same thread. All of the other relations come in "standard" and "external" versions, denoted with a final e. For example $\mathsf{rfe} \triangleq \{(w, r) \mid (w, r) \in \mathsf{rf} \wedge \mathsf{thread}(w) \neq \mathsf{thread}(r)\}$ relates reads that see writes from a different thread.

The first three relations are execution specific. The remaining relations are defined by the architecture. Additionally, the axioms use the following derived relations:

- $\mathsf{fr} \triangleq \{(r, w_1) \mid \exists w_0.(w_0, r) \in \mathsf{rf} \wedge (w_0, w_1) \in \mathsf{co}\}$, pronounced "from-read". Here r is a read, and w_1 is a write which must come after r, since r has seen a write that preceded w_1 on the same location.
- $\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}$ defines "happens-before", ordering events that are causally related.

Various architectures can be defined by instantiating these relations in different ways. On TSO, ppo removes the order between a write and a subsequent read. Thus TSO is defined by setting $\mathsf{ppo} = \mathsf{po} \backslash \mathsf{WR}$, $\mathsf{fences} = \mathsf{mfence}$, $\mathsf{prop} = \mathsf{ppo} \cup \mathsf{rfe} \cup \mathsf{fr}$, where WR is the set of all write-read pairs and mfence orders all memory events before a fence with respect to those after the fence.

Executions must satisfy several sanity conditions. For example, co must be a partial order relating only writes to the same location, which is a total order per location. In addition rf is a relation matching each read to a write with the same value and location. For emphasis, we refer to executions that fulfil these requirements as *sane*.

A sane execution is *valid* if it satisfies the four *AMT axioms*. By (No-Thin-Air), causality cannot be cyclic. By (SC-Per-Location), each location taken separately is sc, where $\mathsf{po\text{-}loc}$ is po restricted to events on the same location. By (Observation), events hidden in the causal past cannot be observed. By (Propagation), writes must be propagated in an order consistent with coherence.

$$\mathsf{acyclic}(\mathsf{hb}) \qquad \text{(No-Thin-Air)}$$

$$\mathsf{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}) \qquad \text{(SC-Per-Location)}$$

$$\mathsf{irreflexive}(\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*) \qquad \text{(Observation)}$$

$$\mathsf{acyclic}(\mathsf{co} \cup \mathsf{prop}) \qquad \text{(Propagation)}$$

We write relations using both set and arrow notation; thus $(a, b) \in \mathsf{po}$ is synonymous with $a \xrightarrow{\mathsf{po}} b$. We also pun between events and their labels in examples.

3 Linearisability for Weak Memory

The goal of the paper is to distinguish "good" implementations, those that ensure contextual refinement, from "bad" ones, that do not. In this section, we present some examples that motivate real-time and causal hb-linearisability as well as the contextual refinement and compositionality properties that they must guarantee.

3.1 Real-time hb-linearisability

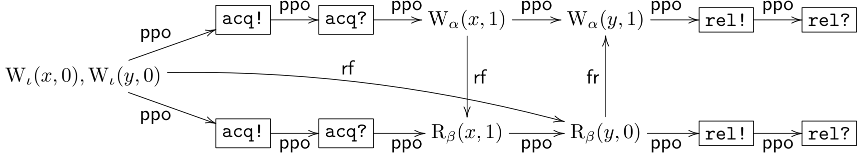
Consider the following code, which uses a lock to coordinate the activities of two threads.

```
Init: x, y = 0, 0
Thread α: lock.acq(); x := 1; y := 1; lock.rel();
Thread β: lock.acq(); print "x=" x; print "y=" y; lock.rel();
```

The abstract specification for a lock will forbid two returns from acq without an intervening call to rel. From this, a programmer would expect that any execution of the client running against a correctly implemented lock must print y=1 whenever it prints x=1.

For sc-memory, one can establish the validity of this reasoning using classical linearisability. This form of reasoning is unsound, however, for relaxed memory.

To see this, consider the following example, which is a possible execution of the program using the abstract lock specification executing under TSO memory. We extend the AMT model to include events denoting method invocation and response. The invocation of the acquire method is depicted as acq!. The return of the acquire method is depicted as acq?. The release method is similar.



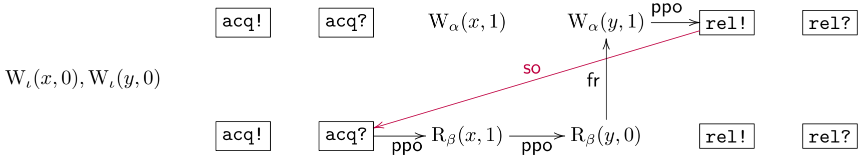
We show the threads in two rows, with events ordered left to right in program order. Applying AMT's framework results in the set of ppo, rf and fr edges between memory events shown above. Since AMT do not address abstract method calls, there are no edges into and out of invocation and response events, except ppo. TSO only relaxes program order between a write and subsequent read. Hence, in the execution above, all program order between memory events is preserved. Here, we have also preserved program order into and out of the method events — in the next subsection and throughout the paper, we consider other strategies for handling the interaction of memory and method events.

The execution satisfies all four AMT axioms, and hence is accepted as a valid execution. However, it clearly violates a programmer's intuition since it describes an execution that prints x=1 and then y=0.

Of course the problem here is that the abstract method calls are being ignored by the memory-model axioms. To remedy this, we must introduce additional

happens-before order that is derived from the abstract specification.[6] We specify this using a *specification order*, so, which relates each invocation to the responses that must *happen-after* that invocation.

If `rel!` in thread α occurs before `acq?` in thread β, the programmer should be able to assume that any event that precedes `rel!` (in program order) must happen before any event that follows the corresponding `acq?`. This should invalidate the execution above. We repeat that execution below, augmented with so, showing only the edges relevant to invalidating this execution.



With hb extended to include so, the exhibited cycle above contradicts AMT's Observation axiom, assuming ppo between a memory and method event is included in prop. Thus, the execution is considered invalid, as desired.

In Section 4, we formalise the concept of a specification augmented with happens-before information and describe its effect on a program execution. We also define real-time hb-linearisability, which is an extension of linearisability that allows one to distinguish a good implementation of an augmented specification from a bad one. A good implementation of the lock must be able *guarantee* the happens-before relation required by the specification. To reason about such implementations, we must also enrich the semantics of implementations to relate method invocation/response events with memory events.

3.2 Causal hb-linearisability

Real-time hb-linearisability is appropriate for tightly coupled systems, such as current generation multicore processors. In distributed systems, however, the cost of real-time synchronisation is high, making real-time hb-linearisability unattractive. We propose *causal* hb*-linearisability* as an alternative notion that requires less synchronisation overhead. Causal hb-linearisability may be appropriate for future generation multicore processors: as the number of cores increases, the necessary synchronisation overhead may force these systems to adopt a looser model (c.f. [28]).

Linearisability requires that the order of non-overlapping methods be preserved in the specification. In the literature, this constraint is motivated by showing that compositionality fails if one requires only that the order of method calls in each thread be preserved. We reexamine these examples in order to motivate causal hb-linearisability.

[6] In fact, APIs such as `java.util.concurrent` document the happens-before behaviour of the methods using edges from the beginning of one method activation to the end of another (or a set of others); that is, from call to return.

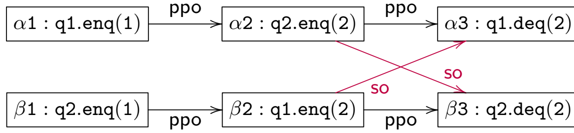


Fig. 1. Non-compositional execution

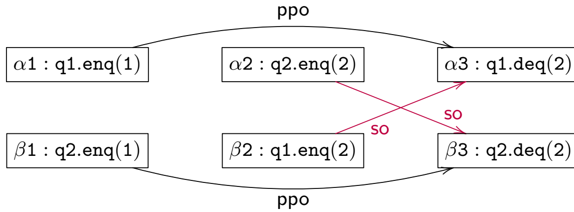


Fig. 2. Compositional execution

Fig. 1 shows a well-known example [18], with method invocation and response collapsed into a single atomic event (shown within a box) for simplicity. Here, threads α and β interact via a pair of queues. The queue specification naturally imposes hb order between an enqueue and a subsequent dequeue of the same element; therefore, the figure shows so edges between $\alpha 2$ and $\beta 3$, as well as between $\beta 2$ and $\alpha 3$. In addition, the figure shows the preserved program order (ppo) between the calls on the two queues in each thread. Recall from Section 2 that $\mathsf{ppo} \subseteq \mathsf{hb}$.

If we consider either q1 and q2 in isolation, the execution is linearisable, since the second enqueue operation for each queue can be considered to have taken effect first. However, the order for each queue is impossible given the order *between* queues. In particular, due to the hb edges when restricting the execution to a single queue and the FIFO ordering requirement of a queue specification, the order of operations for q1 must be $\beta 2 \xrightarrow{\mathsf{hb}} \alpha 1 \xrightarrow{\mathsf{hb}} \alpha 3$, while the order for q2 must be $\alpha 2 \xrightarrow{\mathsf{hb}} \beta 1 \xrightarrow{\mathsf{hb}} \beta 3$. In the full trace, we get a cycle $\alpha 1 \xrightarrow{\mathsf{hb}} \alpha 2 \xrightarrow{\mathsf{hb}} \beta 1 \xrightarrow{\mathsf{hb}} \beta 2 \xrightarrow{\mathsf{hb}} \alpha 1$.

Herlihy and Wing solve this problem by *strengthening* the definition to require that linearisability preserve real-time order of non-overlapping method calls. Thus, the execution of at least one queue in Fig. 1 must be invalidated.

An alternative is to *weaken* ppo to remove the order between events on independent queues. This is analogous to the way events on independent variables are handled under the ARM and Power memory models. Note that we are free to make such a choice here, outside of the implementation memory model, since the ppo order here is at the abstract level between method events. The result is shown in Fig. 2. Here, $\alpha 1$ and $\alpha 3$ are hb ordered, but $\alpha 2$ is not ordered with respect to either event. In this case composition holds. We formalise this intuition as *causal* hb-*linearisability* in Section 4.

We prove abstraction for both real-time and causal hb-linearisability. The story for composition is more complex since clients may be obtrusive, enforcing additional program order between events on different objects. Formally, an execution is *unobtrusive* if it matches a specification string v such that whenever $v = s \cdot a! \cdot a? \cdot t \cdot b! \cdot b? \cdot u$ and $a? \xrightarrow{\mathsf{hb}} b!$ in the execution then $a! \xrightarrow{\mathsf{so}} b?$ in the specification. A client is unobtrusive if all of its executions are unobtrusive. An obtrusive client, such as the one in Fig. 1, may place a fence between the method calls, or use some form of synchronisation to enforce order between them, for example by writing-to and then reading-from another thread. An unobtrusive client, such as the one in Fig. 2, must perform no such synchronisation. We show that causal hb-linearisability satisfies compositionality if the client is unobtrusive.

Obtrusive clients are not problematic if the specification is *commutative*, i.e., if for any specification string $s \cdot a! \cdot a? \cdot t \cdot b! \cdot b? \cdot u$ either $a! \xrightarrow{\mathsf{so}} b?$ or $s \cdot b! \cdot b? \cdot t \cdot a! \cdot a? \cdot u$ is a specification string. Figs. 1 and 2 show the interaction of a client with a composite double-queue. A double-queue specification is not commutative because it does not permit reordering of calls to `enq`, yet the usual specification does not contain a happens-before specification among calls to `enq`. An example of a commutative specification is a double *bag* or *multi-set* where a call to `add` is specified to happen-before the corresponding `remove`, but where all commutations are permitted between the operations on separate elements. We show that causal hb-linearisability satisfies compositionality if the specification is commutative.

4 Traces and Weak-Memory Semantics

In this section we formalise the interaction between a *client* and a set of *objects*.

We divide the event set $\mathbb{E}$ into four disjoint subsets: Let $\mathbb{C}$ be the set of *client events*, let $\mathbb{O}$ be the set of *object events*, let $\mathbb{I}$ be the set of *invocation events* and let $\mathbb{R}$ be the set of *response events*. We use $\mathbb{M} \subseteq \mathbb{I} \cup \mathbb{R}$ to range over *method events*. Like others [15–17], we assume clients and objects only communicate via the object interface, specified as subset of $\mathbb{I} \cup \mathbb{R}$. Thus, clients and objects must operate over disjoint sets of locations: $\mathsf{location}(e) \neq \mathsf{location}(f)$ for any $e \in \mathbb{C}$ and $f \in \mathbb{O}$.

For any relation $R \subseteq \mathbb{E} \times \mathbb{E}$ and set $\mathbb{X} \subseteq \mathbb{E}$, let $R|\mathbb{X}$ denote the restriction of R to $\mathbb{X}$, i.e., $R|\mathbb{X} = R \cap (\mathbb{X} \times \mathbb{X})$.

In order to connect AMT-style executions to specifications, we work with strings of events, which we refer to as *traces*[7]. While our definitions are given directly in terms of traces, we often use program syntax in examples. It is straightforward to define a semantics which gives the denotation of programs as sets of traces, where memory reads and method returns may yield any value. For example, the semantics of "`x:=1;push(5)`" is the set $\{W_\alpha(x, 1) \cdot push!_\alpha(5) \cdot push?_\alpha \mid \alpha \in Threads\}$. The semantics of "`r1:=pop();r2:=x`" is the set $\{pop!_\beta \cdot$

[7] Since events include unique identifiers (and therefore cannot repeat), there is an isomorphism between strings of events and total orders over finite set of events.

$pop?_\beta(u) \cdot \mathrm{R}_\beta(x, v) \mid \beta \in \mathit{Threads}, u \in \mathit{Values}, v \in \mathit{Values}\}$. The semantics of "`x:=1;push(5) || r1:=pop();r2:=x`" is any interleaving of these where $\alpha \neq \beta$. Note that both the pop and the read of `x` may return any value.

In the remainder of this introductory text, we consider method events and memory events independently: method events relate to specifications and memory events relate to executions. In the following subsections, we show how these can be combined, both for abstract and concrete object systems.

First we discuss method events. Rather than modelling specifications using formal languages such as Larch, Z, or LTL, we model specifications semantically as sets of strings of method events. Strings provide a total order on the method events, which is sufficient to capture sequential behaviours. When considering event strings as specifications, we ignore the thread identifier in events. For example, the specification of stack includes strings such as $push!(5) \cdot push? \cdot pop! \cdot pop?(5)$. Thus a trace of method events may be seen directly as an element of a specification, where we ignore thread identifiers. For example, the trace $push!_\alpha(5) \cdot push?_\alpha \cdot pop!_\beta \cdot pop?_\beta(5)$ is a valid trace for a stack, whereas neither $push!_\alpha(5) \cdot push?_\alpha \cdot pop!_\beta \cdot pop?_\beta(1)$ nor $pop!_\beta \cdot pop?_\beta(5) \cdot push!_\alpha(5) \cdot push?_\alpha$ is valid.

We now discuss memory events. From a trace, we can generate AMT executions as follows.

Definition 1. *A tuple* $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop})$ *is an* execution *of trace t if these relations satisfy AMT's sanity conditions (see Section 2), where program order is given by* $\mathsf{po}_t = \{(e, f) \mid (e, f) \in t \wedge \mathsf{thread}(e) = \mathsf{thread}(f)\}$.

Let $\mathsf{execs}(t)$ *be the set of executions of t. We use τ to range over executions.*

Note that we require executions to be sane, but do not enforce validity at this stage. We discuss validity in the following subsections. We usually drop subscripts from order relations, preferring po to po_t, etc.

A single trace may give rise to many executions. For example, the program "`r=x || x:=5 || x:=5`" gives rise to a set of traces which includes $\mathrm{R}_\alpha(x, 5) \cdot \mathrm{W}_\beta(x, 5) \cdot \mathrm{W}_\gamma(x, 5)$. Executions of this trace may have either $\mathrm{W}_\beta(x, 5) \xrightarrow{\mathsf{rf}} \mathrm{R}_\alpha(x, 5)$ or $\mathrm{W}_\gamma(x, 5) \xrightarrow{\mathsf{rf}} \mathrm{R}_\alpha(x, 5)$. This program also gives rise to traces such as $\mathrm{R}_\alpha(x, 1) \cdot \mathrm{W}_\beta(x, 5) \cdot \mathrm{W}_\gamma(x, 5)$, which has no executions, since the read can not be fulfilled by any write.

The trace order of events from the same thread determines program order; however, the trace order between events from different threads is ignored. If $\alpha \neq \beta$, then the traces $\mathrm{R}_\alpha(x, 5) \cdot \mathrm{W}_\beta(x, 5)$ and $\mathrm{W}_\beta(x, 5) \cdot \mathrm{R}_\alpha(x, 5)$ generate exactly the same executions, modulo the trace itself, as do $push!_\alpha(5) \cdot push?_\alpha \cdot pop!_\beta \cdot pop?_\beta(5)$ and $pop!_\beta \cdot pop?_\beta(5) \cdot push!_\alpha(5) \cdot push?_\alpha$.

4.1 Clients with object specifications

We now discuss the semantics of client-object systems, where object behaviours are described by a specification. In terms of condition (†) from the introduction, this section formalises the behaviours of $C[AS]$. As discussed in Section 3, it

is important for specifications to provide happens-before guarantees to client programs to enable writes to propagate in the correct order.

Example 2. Consider the program "`x:=5;push(5) || r1:=pop();r2:=x`". If variable `x` is initialised to 0, then the following is a trace of this program:

$$\mathrm{W}_\iota(x,0)\cdot \mathrm{W}_\alpha(x,5)\cdot \mathit{push}!_\alpha(5)\cdot \mathit{push}?_\alpha \cdot \mathit{pop}!_\beta \cdot \mathit{pop}?_\beta(5)\cdot \mathrm{R}_\beta(x,0)$$

There are valid AMT executions of this trace. Here the thread β returns a value 0 for x, missing the value 5 written by thread α, even if we assume that the memory model guarantees $\mathrm{W}_\iota(x,0) \xrightarrow{\mathsf{hb}} \mathrm{W}_\alpha(x,5)$. We wish to disallow such traces via extra happens-before orders introduced via the stack specification. In particular, we enhance the specification with an additional ordering relation that ensures each *push* is hb ordered before the corresponding *pop*. When used in a client program, we assume that such an enhanced specification induces additional order, namely that it ensures $\mathrm{W}_\alpha(x,5) \xrightarrow{\mathsf{hb}} \mathrm{R}_\beta(x,0)$. Now, if the memory model ensures $\mathrm{W}_\iota(x,0) \xrightarrow{\mathsf{hb}} \mathrm{W}_\alpha(x,5)$ the trace becomes invalid, as intended. □

In order to encode happens-before information, we take a specification string to be pair consisting of a string of method events, h, and a *specification-based happens-before order* so, relating events in h. For example, in the stack specification $h = \mathit{push}!(5)\cdot \mathit{push}?\cdot \mathit{pop}!\cdot \mathit{pop}?(5)$, we expect that $\mathit{push}!(5) \xrightarrow{\mathsf{so}} \mathit{pop}?(5)$. Various choices of so are possible. The Java concurrency APIs specify that each push happens-before the corresponding pop. Many concurrent implementations actually give stronger guarantees, which could be included in the specification if one wished. For example, a Trieber stack guarantees that a push happens-before the matching pop *and* every subsequent pop. If this specification were adopted, the client programmer would be able to make stronger assumptions. Our results are parametrised by a chosen specification.

In the following definition, we recall that $\mathbb{M} \subseteq \mathbb{I} \cup \mathbb{R}$ is the set of method events, and require that so only relate invocations to responses. In addition, so must be consistent with h.

Definition 3 (Specification). *A* specification *is a pair* $(\mathbb{M}, H)$*, where* $H \subseteq 2^{\mathbb{M}\times\mathbb{M}} \times 2^{\mathbb{I}\times\mathbb{R}}$ *such that for each* $(h, \mathsf{so}) \in H$*,* h *is a total order and* $\mathsf{so} \subseteq h$*.*

We now define what it means for a client to interact with an abstract specification: When projected to client events, we must have a sane AMT execution. When projected to method events, we must have a specification string.

Definition 4 (Client-specification execution). *Let* $AS = (\mathbb{M}, H)$ *be a specification,* C *be a client and* $t \in (\mathbb{M}\cup\mathbb{C})^*$ *be a trace of* $C[AS]$*. We say that the tuple* $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop})$ *is a* client-specification execution for so *iff*

- $(t|\mathbb{C}, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}) \in \mathsf{execs}(t|\mathbb{C})$*, where* $t|\mathbb{C}$ *is the trace* t *restricted to elements in* $\mathbb{C}$*, and*
- $\mathsf{so} \subseteq \mathbb{I}\times\mathbb{R}$ *is an order such that* $(t|\mathbb{M}, \mathsf{so}) \in H$*.*

A valid client-specification execution must satisfy the AMT axioms, where method events are included in the happens-before relation by lifting so to relate memory events ordered by $\mathsf{po};\mathsf{so};\mathsf{po}$.

Definition 5 (Valid client-specification execution). *A* client-specification execution for so *is* valid *iff the AMT axioms hold with* $\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{hbs}$*, where*

$$\mathsf{hbs} = \{(e, e') \subseteq \mathbb{C} \times \mathbb{C} \mid \exists i \in \mathbb{I}, r \in \mathbb{R}.\ e \xrightarrow{\mathsf{po}}_t i \wedge i \xrightarrow{\mathsf{so}} r \wedge r \xrightarrow{\mathsf{po}}_t e'\}.$$

4.2 Client-implementation traces

We now describe the meaning of a client that executes with an implementation object. In terms of condition (†) from the introduction, this section formalises the behaviours of $C[CS]$. One can interpret a client interaction with a concrete object system as an execution by simply removing method events. For example, if s is the sequence of memory events implementing push, and t is the sequence of memory events implementing pop, then the following is a concrete trace of the program in Example 2:

$$\mathrm{W}_\iota(x, 0) \cdot \mathrm{W}_\alpha(x, 5) \cdot \mathit{push}!_\alpha(5) \cdot s \cdot \mathit{push}?_\alpha \cdot \mathit{pop}!_\beta \cdot t \cdot \mathit{pop}?_\beta(5) \cdot \mathrm{R}_\beta(x, 0).$$

One could say that this trace is valid exactly if $\mathrm{W}_\iota(x, 0) \cdot \mathrm{W}_\alpha(x, 5) \cdot s \cdot t \cdot \mathrm{R}_\beta(x, 0)$ is valid, i.e., the trace with method events removed. While sufficient for some purposes, any connection with the abstract object system is lost. In this section we describe how to integrate method actions into concrete executions so as to support a notion of operational refinement between concrete and abstract systems.

In general, a terminating thread of a client/object interaction has the form $\mathbb{C}^*(\mathbb{I}\mathbb{O}^*\mathbb{R}\mathbb{C}^*)^*$: the client may perform memory actions in $\mathbb{C}$ until it invokes a method, giving control to the object; the object then may perform memory actions $\mathbb{O}$ until it returns, giving control back to the client[8]. In executions of concrete traces, method events are placeholders, which should have no memory effects themselves. Any memory effects should arise from the concrete implementation. Thus we expect that empty methods should have no effects in a concrete system. More generally, our definition should support method inlining.

The problem boils down to which po edges between method events and memory events should be preserved in ppo, and therefore hb. Since method events denote the boundary between client events and object events, there are two sets of edges to consider:

(**1**) po edges between $\mathbb{M}$ and $\mathbb{C}$, and
(**2**) po edges between $\mathbb{M}$ and $\mathbb{O}$.

We cannot preserve *both* (**1**) and (**2**). To see why, consider the trace t below, where c, $c' \in \mathbb{C}$, $i \in \mathbb{I}$, $r \in \mathbb{R}$ and o, $o' \in \mathbb{O}$ are events of the same thread:

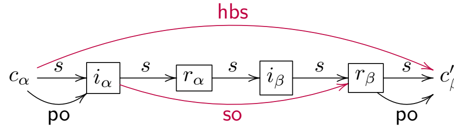
$$c \xrightarrow{t} \boxed{i} \xrightarrow{t} o \xrightarrow{t} \ldots o' \xrightarrow{t} \boxed{r} \xrightarrow{t} c'$$

[8] Recall from the beginning of this section $\mathbb{O}$ and $\mathbb{C}$ range over disjoint sets of memory locations.

The preserved po edges must *not* introduce any new ppo order between c and o that is not present in the memory model since the invocation i, in isolation, cannot affect memory. In other words, if both $c \xrightarrow{\mathsf{po}} i$ and $i \xrightarrow{\mathsf{po}} o$ were preserved, this would ultimately create a transitive happens-before edge between c and o, disallowing them from being reordered even in a memory model that doesn't enforce this restriction. For example, in TSO, we may have $c = W(z, 1)$ and $o = R(x, 2)$, which may be reordered; introduction of a method invocation between c and o should not prevent the reordering from occurring.

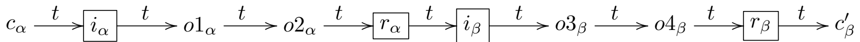
We must also preserve (**1**) and (**2**) in such a way that we are able to decouple object correctness (hb-linearisability) from contextual refinement. Our solution is to *always* preserve (**1**), resulting in a set of edges cio (client-interface order), and *conditionally* preserve (**2**), resulting in a set of edges oio (object-interface order). The intention is to introduce both cio and oio into an extended hb ordering.

To justify our choices, consider the abstract trace s, given below, which shows a client interacting with an abstract specification object. The actions c_α and c_β are client actions of separate threads, α and β.



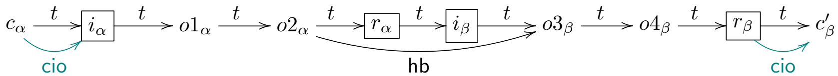
Assume that the specification requires so between i_α and r_β. Thus, by Definition 5, for any execution of s, we must have an hbs edge between client events c_α and c'_β. This (again by Definition 5) means that we have $c_\alpha \xrightarrow{\mathsf{hb}} c'_\beta$ since $\mathsf{hbs} \subseteq \mathsf{hb}$.

Suppose we wish to determine whether the trace t below is a *contextual refinement* of s.



Among other things, we must be able to guarantee $c_\alpha \xrightarrow{\mathsf{hb}} c'_\beta$ for any execution of t (see Definition 9) since this order is present in the specification.

An implementation of the sequential object can take us part of the way there by ensuring hb between object events. Suppose for our example that the implementation guarantees $o2_\alpha \xrightarrow{\mathsf{hb}} o3_\beta$. This, together with the fact that we always preserve (**1**), results in an execution of the following form:

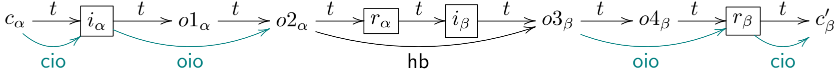


Note that the hb above is introduced via AMT's standard conditions described in Section 2, i.e., without taking invocations and responses into account.

To complete the hb chain from c_α to c'_β, we require edges from i_α to $o2_\alpha$ and from $o3_\beta$ to r_β. The condition for preserving (**2**) is as follows. Suppose o is an object event. We preserve program order from an invocation i to o if after replacing the action in i with an *arbitrarily chosen* memory action to obtain an event i', the relation $\mathsf{ppo} \cup \mathsf{fences}$ orders i' before o. The case for ordering o

before a response event is similar. Since an arbitrary memory action is ordered before o, any client memory action that occurs before i in t must also be ordered with respect to o. Moreover, these conditions are independent of any specific client, and hence our treatment allows one to reason about the properties of the concurrent object (e.g., hb-linearisability) in isolation, relying on our abstraction theorem to guarantee contextual refinement.

Suppose for our example that $o2_\alpha$ and $o3_\beta$ do indeed satisfy the conditions described above. Our execution thus becomes:



The presence of the oio edge to $o2_\beta$ means that any client memory event of α that precedes i_α in program order must also be ordered with $o2_\beta$ (since an arbitrary action was considered when constructing oio). Since we do not have an oio edge to $o1_\alpha$, it would be possible to reorder c_α with $o1_\alpha$ if the memory model semantics permits the reordering. (The same applies to $o4_\beta$ and c'_β.) Thus, we have only introduced as much order as necessary.

Example 6. Consider TSO, and suppose $c_\alpha = W_\alpha(x, 1)$, $o1_\alpha = R_\alpha(y, 2)$ and $o2_\alpha = W_\alpha(z, 4)$. We do not have $i_\alpha \xrightarrow{\mathsf{oio}} o1_\alpha$ since for TSO, $\mathsf{ppo} = \mathsf{po} \backslash \mathsf{WR}$. However, we do have $i_\alpha \xrightarrow{\mathsf{oio}} o2_\alpha$. The program under TSO could reorder c_α and $o1_\alpha$, but would never reorder c_α and $o2_\alpha$. Now suppose $o1_\alpha = \mathsf{fence}$ and $o2_\alpha = R_\alpha(y, 3)$. Again, we have $i_\alpha \xrightarrow{\mathsf{oio}} o2_\alpha$, but in this instance, the order from i_α is generated by the fence.

We now formalise both orders in the context of an *extended execution*, i.e., an execution extended with orders cio and oio. Such a definition is necessary because the definition of oio requires relabelling of invocation actions within a trace. In the definition below, we let $\mathsf{relabel}(e, a)$ denote the event e with its action relabelled to a, $\mathsf{labels}(e) = \{\mathsf{relabel}(e, a) \mid a \text{ is a memory action}\}$ denote the set of all possible relabelings of e and $t[e'/e]$ denote the trace t with event e replaced by e'.

Definition 7 (Client-implementation execution). *For a trace t, we say that the tuple $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}, \mathsf{cio}, \mathsf{oio})$ is a* client-implementation execution *iff the $(t|(\mathbb{C} \cup \mathbb{O}), \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}) \in \mathsf{execs}(t|(\mathbb{C} \cup \mathbb{O}))$, and*

$$\mathsf{cio} = \mathsf{po}_t \cap ((\mathbb{C} \times \mathbb{I}) \cup (\mathbb{R} \times \mathbb{C}))$$
$$\mathsf{oio} = \{(i, o) \in \mathbb{I} \times \mathbb{O} \mid \mathsf{IO}(t, i, o)\} \cup \{(o, r) \in \mathbb{O} \times \mathbb{R} \mid \mathsf{OR}(t, o, r)\}$$

where IO *and* OR *are defined as follows.*

$$\mathsf{IO}(t, i, o) = \forall i' \in \mathsf{labels}(i).\ \forall \tau \in \mathsf{execs}(t[i'/i] \mid (\mathbb{C} \cup \mathbb{O} \cup \{i'\})).\ i' \xrightarrow{\mathsf{ppo} \cup \mathsf{fences}}_\tau o$$
$$\mathsf{OR}(t, o, r) = \forall r' \in \mathsf{labels}(r).\ \forall \tau \in \mathsf{execs}(t[r'/r] \mid (\mathbb{C} \cup \mathbb{O} \cup \{r'\})).\ o \xrightarrow{\mathsf{ppo} \cup \mathsf{fences}}_\tau r'$$

In this definition, $e \xrightarrow{\mathsf{ppo} \cup \mathsf{fences}}_\tau f$ denotes $(e, f) \in \mathsf{ppo}_\tau \cup \mathsf{fences}_\tau$, recalling that ppo_τ and fences_τ are the ppo and fences relations of the execution τ, respectively.

Within $\mathsf{IO}(t, i, o)$, for any i' obtained by replacing the action in i with a memory action, and any execution τ of the trace t with i replaced by i' restricted to $\mathbb{C} \cup \mathbb{O} \cup \{i'\}$, we have that i' is ordered before o with respect to ppo_τ or fences_τ. The predicate $\mathsf{OR}(t, i, o)$ is similar.

Definition 8 (Valid client-implementation execution). *We say client-implementation execution is* valid *iff the AMT axioms hold, where* $\mathsf{hb} \triangleq \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe} \cup \mathsf{cio} \cup \mathsf{oio}$.

Our notion of contextual refinement is based purely on the observations that a client makes over the memory and object states. Thus, it simply ensures that every valid execution of the client when using the implementation object is a possible execution of the client when it uses the specification object.

Definition 9 (Contextual refinement). *Suppose t is a trace of $C[CS]$ and s is a trace of $C[AS]$. We say t* contextually refines *s (denoted $s \sqsubseteq t$) iff*

- $t|\mathbb{C} = s|\mathbb{C}$,
- *whenever* $(t, \mathsf{co}, \mathsf{rf}, \mathsf{ppo}, \mathsf{fences}, \mathsf{prop}, \mathsf{cio}, \mathsf{oio})$ *is a valid client-implementation execution,* $(s, \mathsf{co}|\mathbb{C}, \mathsf{rf}|\mathbb{C}, \mathsf{ppo}|\mathbb{C}, \mathsf{fences}|\mathbb{C}, \mathsf{prop}|\mathbb{C})$ *is a valid client-specification execution.*

The first condition requires that s and t restricted to client events are equal (i.e., they have the same denotational behaviour), whereas the second requires that a valid execution of t can be restricted to form a valid execution of s. In particular, if t is valid, then s must also be valid.

Contextual refinement is lifted to the level of programs in the standard manner. We say $C[AS]$ is *contextually refined* by $C[CS]$, denoted $C[AS] \sqsubseteq C[CS]$, iff for every valid trace t of $C[CS]$, there exists a valid trace s of $C[AS]$ such that $s \sqsubseteq t$. We say AS is *contextually refined* by CS, denoted $AS \sqsubseteq CS$ iff for any client C, we have $C[AS] \sqsubseteq C[CS]$.

4.3 Implementation objects and happens-before linearisability

In this section, we formalise the correctness expectations on an implementation object in terms of a sequential specification. The notions we develop are based on linearisability. In the context of weak memory, we show that linearisability is not sufficient: additional requirements must be enforced. At the same time, weak memory makes it natural to look at notions of linearisability that do not strictly enforce realtime order. In terms of (†), this section formalises the sorts of behaviours CS must satisfy in order to prove the abstraction property.

Linearisability in a weak memory setting must preserve the happens-before order of an abstract specification. An implementation trace comprises client/object memory events as well as invocation/response events of object operations. From the perspective of an object, invocation/response events abstractly represent a client's memory events in program order. Thus preserved program order between object memory events and invocation/response events are execution specific, and introduced into the happens-before order of a trace.

The final component of hb-linearisability is a restriction on how invocations and responses can be (re)ordered. In a weak memory setting there is more than one potential restriction. Two of these are to: (a) order operation calls according to their real-time program order (real-time hb-linearisability), and (b) order operation calls according to their happens-before order (causal hb-linearisability). Choice (a) is closer to Herlihy and Wing's original definition, while (b) is closer to an ordering one might expect in a weak-memory setting.

In the definition below, like linearisability [19], since operations may take effect before they return, we allow histories to be extended by adding matching responses to operations that have been invoked but not yet returned.

Definition 10. *A valid client execution is* real-time hb-linearisable *with respect to* (h, so) *iff it can be extended to an execution of some trace* t *with happens-before relation* hb *such that the following holds:*

$$\forall \alpha \in \mathit{Threads}.\ [t|\alpha|(\mathbb{I} \cup \mathbb{R}) = h|\alpha] \vee [\exists i \in \mathbb{I}.\ t|\alpha|(\mathbb{I} \cup \mathbb{R}) = (h|\alpha) \cdot i] \qquad \text{(Permutation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{t} i \Rightarrow r \xrightarrow{h} i \qquad \text{(RTO-Preservation)}$$

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ i \xrightarrow{\mathsf{so}} r \Rightarrow i \xrightarrow{\mathsf{hb}} r \qquad \text{(HB-Satisfaction)}$$

An execution is linearisable with respect to a specification $AS = (\mathbb{M}, H)$ *if it is linearisable for some* $(h, \mathsf{so}) \in H$*.*

Conditions (Permutation) and (RTO-Preservation) are equivalent to Herlihy and Wing's original requirements for linearisability [19]. Thus, hb-linearisability implies standard linearisability. Condition (HB-Satisfaction) ensures that the order between invocations and responses (of different operations) expected by the specification is respected by the happens-before order in the implementation.

Definition 11. Causal hb-linearisability *differs from real-time* hb*-linearisability only in that condition* (RTO-Preservation) *is replaced by:*

$$\forall i \in \mathbb{I}, r \in \mathbb{R}.\ r \xrightarrow{\mathsf{hb}^+} i \Rightarrow r \xrightarrow{h} i \qquad \text{(HB-Preservation)}$$

Real-time and causal hb-linearisability are incomparable since po and hb^+ are incomparable. The differences between these requirements are shown in Figs. 3-6. In Fig. 3, the execution is considered to be sequential according to both conditions, but in the second example, the method calls are causally ordered in a different order to real-time order. In Fig. 4, both executions are concurrent according to real-time order, but sequential according to causal order, in Fig. 5 the operations are considered to be concurrent according to both orders, and in Fig. 6 the execution is real-time sequential, but causally concurrent.

Consider the following simplified version of the queue example from Section 3.2.

Example 12. A two-place buffer has operations $\mathtt{put}_1$, $\mathtt{put}_2$, $\mathtt{get}_1$ and $\mathtt{get}_2$. The sequential specification states that a call to $\mathtt{get}_i$ must return the argument given on the most recent call to $\mathtt{put}_i$. If we follow the model of the data structures in `java.util.concurrent`, the expected happens-before relation is that $\mathtt{put}_i$

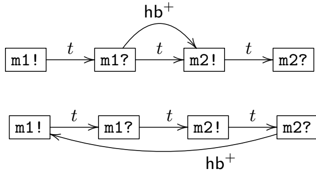


Fig. 3. Real-time and causal sequential

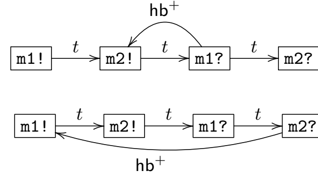


Fig. 4. Real-time concurrent, causal sequential

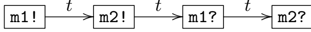


Fig. 5. Real-time and causal concurrent

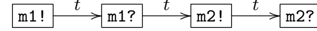


Fig. 6. Real-time sequential, causal concurrent

happens-before any $\mathtt{get}_i$ that returns a matching value. In particular, note that there is no happens-before expectation between $\mathtt{put}_1$ and $\mathtt{get}_2$.

It is possible to implement the two-place buffer using independent synchronisation variables. Supposing that the buffer initially holds zeros, the client

Thread α: $\mathtt{put}_1\mathtt{(5);}\ \mathtt{get}_2\mathtt{()}$
Thread β: $\mathtt{put}_2\mathtt{(5);}\ \mathtt{get}_1\mathtt{()}$

can return zero for both calls to `get`. This execution is correct with respect to causal hb-linearisability, but not with respect to real-time hb-linearisability.

We now turn our attention to the cases when the two notions coincide: Real-time hb-linearisability and causal hb-linearisability coincide if for the execution $\tau \in \mathsf{execs}(t)$ under consideration $\mathsf{hb}^+ = t$. In particular, for the sc memory model, real-time and causal hb-linearisability coincide.

We simply use the term hb-linearisability whenever we do not distinguish between real-time and causal hb-linearisability. The next definition lifts hb-linearisability to the level of objects in the standard manner.

Definition 13 (hb-linearisable implementation). *We say that an object CS is an* hb-linearisable implementation *of specification $AS = (\mathbb{M}, H)$ if for all clients C, and all valid executions τ of $C[CS]$, τ is* hb*-linearisable with respect to some $(h, \mathsf{so}) \in H$.*

4.4 Establishing hb-linearisability

In this section, we demonstrate that for some implementations it is no more difficult to establish hb-linearisability than it is to establish standard linearisability. Of course, establishing standard linearisability on a relaxed memory model is still more difficult than under sc memory.

For example, in the Treiber stack algorithm, each method call must perform a compare-and-set operation on a single memory location, representing the top of the stack. The order of successful CAS operations is the linearisation order

used to establish linearisability. It also establishes happens-before between the call of a method and the return of every method that follows it in linearisation order, assuming that CAS is given acquire/release semantics, as in Java and TSO.

We can establish a similar result for any classically linearisable implementation of a collection class, under one of two assumptions:

- there is a memory fence at beginning of every mutator method and at the end of every accessor method, or
- data values are stored in memory locations with acquire/release semantics.

In the remainder of this section, we establish that, in either case, a classically linearisable collection already satisfies the happens-before requirements of the Java collections API.

A *collection class* refers to common data structures such as Stacks, Queues, Lists, Trees that are containers of elements of objects of a given type. In the rest of this discussion, we pick Stack as the example; however, our discussion applies equally well to the other examples. We use the happens-before semantics of the Java collections classes.

Recall that the signature of a `Stack<T>` of elements of type T is given by `void push(T), T pop(), T top()` and `boolean isEmpty()`. The happens-before requirement is carried by the objects of type `T`, i.e. there is a happens-before edge to the return of any `pop` or `top` from the invocation of the corresponding `push`. Notably, there are no happens-before requirement between different push or different pop methods.

Consider an implementation I of `Stack<T>`. We say that I is *generic* if the operations that it performs on values of type `T` are restricted to load, and store. *All* classical Stack (indeed, collection!) algorithms, such as Treiber stack, follow this discipline. We call such implementations generic because such an implementation treats type `T` as abstract, only writing and reading the values, thus eschewing any operation that exploit the structure of type `T`. The correctness of I is implied by studying the traces that restrict the push methods to be of form `push(new T())`, i.e. every push is of a new object reference that has not been seen thus far[9].

In such a restricted trace, it is immediately clear that there is a location that is written by `push(o)` that is also read by a `pop` or `top` method that returns `o`, since the argument to the push is a new reference. Since the AMT axioms ensure that reads-from is always contained in happens-before, this ensures that

[9] The proof of this fact is inspired by proofs of information flow. In an execution trace of I, consider the locations partitioned into "low" locations $\mathcal{L}$ that store values not of type `T` and "high" locations $\mathcal{H}$ that store values of type `T`. Two memories $(\mathcal{L}_1, \mathcal{H}_1)$ and $(\mathcal{L}_2, \mathcal{H}_2)$ are related by $\mathcal{E}$ if they agree on their "low" parts. By the restrictions on generic implementations, any program statement in a generic implementation preserves the $\mathcal{E}$ relationship of memories. Thus, in order to validate I, it suffices to consider the execution traces where pushes are restricted to have new references as parameters.

any classically linearisable generic implementation of Stack will always have the required happens-before to `pop` or `top` from the corresponding `push`.

5 Abstraction and compositionality

Having established the formal definitions of real-time and causal hb-linearisability, we now turn to their contextual guarantees. That is, we return to our questions originally raised in Sections 1 and 3.

Our first theorem and its associated corollary establishes trace abstraction (or contextual refinement) for (real-time and causal) hb-linearisability. That is, if the implementation object under consideration satisfies hb-linearisability with respect to the corresponding abstract object, any (observable) client behaviour when it uses the implementation is a possible behaviour when it uses the abstract specification.

Theorem 14. *Suppose t is a trace of $C[CS]$. If for any $\tau \in \mathsf{execs}(t)$, τ is* hb-*linearisable with respect to AS, then there exists a valid trace s of $C[AS]$ such that $s \sqsubseteq t$.*

The proof of the theorem amounts to showing that assuming object calls of t are hb-linearisable with respect to an abstract history h then, there is a valid trace t' that is a permutation of t such that: all calls in t' are atomic and in the order given by h.

Corollary 15. *If CS is an* hb-*linearisable implementation of AS, then $AS \sqsubseteq CS$.*

Following Herlihy and Wing, we say hb-linearisability is compositional if two objects that individually satisfy hb-linearisability together satisfy hb-linearisability. For simplicity, we define (and verify) the compositionality property for two objects. This trivially generalises to a composition result for n objects. For causal hb -linearisability, recall the notion of *commutative* specification and *unobtrusive* client from Section 3.2.

Theorem 16. *Let $AS = (\mathbb{I} \cup \mathbb{R}, H)$ be a specification. Suppose $\mathbb{I} = \mathbb{I}_1 \uplus \mathbb{I}_2$ and $\mathbb{R} = \mathbb{R}_1 \uplus \mathbb{R}_2$, such that for each $(h, \mathsf{so}) \in H$, $((\mathbb{R}_2 \times \mathbb{I}_1) \cap \mathsf{so}) = \emptyset \wedge ((\mathbb{R}_1 \times \mathbb{I}_2) \cap \mathsf{so}) = \emptyset$. For $i \in \{1, 2\}$, let $\mathbb{M}_i = \mathbb{I}_i \cup \mathbb{R}_i$ and $AS_i = (\mathbb{M}_i, H|\mathbb{M}_i)$ be the specification restricted to $\mathbb{M}_i$, where $H|\mathbb{M}_i = \{(h|\mathbb{M}_i, \mathsf{so}|\mathbb{M}_i) \mid (h, \mathsf{so}) \in H\}$.*

Consider a trace t of $C[CS_1, CS_2]$ and $\tau \in \mathsf{execs}(t)$.

- *If τ is real-time* hb-*linearisable with respect to each AS_i, then τ is real-time* hb-*linearisable with respect to AS.*
- *If τ is causal* hb-*linearisable with respect to each AS_i and both AS_1 and AS_2 are* commutative*, then τ is causal* hb-*linearisable with respect to AS.*
- *If τ is causal* hb-*linearisable with respect to each AS_i and C is* unobtrusive*, then τ is causal* hb-*linearisable with respect to AS.*

The assumption $((\mathbb{R}_2 \times \mathbb{I}_1) \cap \mathsf{so}) = \emptyset \wedge ((\mathbb{R}_1 \times \mathbb{I}_2) \cap \mathsf{so}) = \emptyset$ ensures that AS can be projected onto two independent objects. It is possible to generalise this by assuming that clients ensure any "cross-object" happens-before requirements in AS. However, such a theorem is more complicated to state formally and hence has been omitted for space reasons.

The consequences of compositionality for real-time hb-linearisability, are similar to those that Herlihy and Wing observed for standard linearisability: we need only add that no so-order is lost when we combine independent suborders.

The results for causal hb-linearisability are less familiar. Consider the commutative specification of a bag, where each `remove` happens-after the corresponding `add`; the second clause of Theorem 16 establishes that two disjoint causal hb-linearisable bags may be combined to produce a "larger" hb-linearisable bag. Next, consider a double queue, as in section 3.2; if we can partition the client into independent, non-synchronising thread groups, such that each group only interacts with a single queue, then the third clause of the theorem tells us that executions of the client with the double queue will be causal hb-linearisable.

6 Conclusion

This paper has developed two modified notions of linearisability for weak memory models based on partially ordered notions of execution. These address the inability of standard linearisability [19] to ensure that programmer expectations about the happens-before relations are met by the objects used (see Section 3). Our work has been integrated with the Alglave et al's (AMT) memory model axioms [3], permitting it to uniformly address a variety of memory models. We enhance the axioms of AMT to address abstract objects, invocation/response events of concrete implementations and the consequent modelling of additional happens-before order for both abstract and implementation levels.

We provide two alternative definitions. Our first extension, real-time hb-linearisability, simply adds an additional condition by requiring that the happens-before requirement of the abstract specification is appropriately reflected by the implementation. The second, causal hb-linearisability, additionally replaces the real-time order preservation property by a happens-before order preservation property. We establish composition and abstraction for our two definitions of linearisability.

The results in this paper advance the state of the art in the following ways: firstly, we obtain a contextual refinement theorem and a composition theorem. Secondly, we build on the framework for memory models created by Alglave et al [3]; so our approach is generic, and applicable to any weak memory model that is encompassed by [3], so we are also able to address TSO, C11 release-acquire, ARM and POWER. Thirdly, our framework permits us to explore both global-time and partial-order time variants of the definition of linearisability.

Related work. Since its introduction by Herlihy and Wing [19], linearisability has emerged as the de-facto criterion for correctness of concurrent objects. We refer

the reader to our survey article [13] for a detailed overview and bibliography of the large amount of research into the verification of linearisability. These investigations were carried out in the context of sequential consistency. Below, we discuss the most closely related papers that explore linearisability in the context of relaxed memory.

The study of linearisability, in the presence of relaxed memory was initiated by Burckhardt et al. [7]. This study was carried out for the TSO memory model. The key idea behind this paper is the association of a separate notion of atomic update of memory to a method call in addition to the usual notion of an atomic execution of the method. Thus, a methods is not atomic in this perspective. In our presentation, the happens before relation in the specification describes the requirements of memory visibility on the implementation. In order to prove hb-linearizability, these requirements have to be established, though we have seen that in some cases this proof is immediate.

A notion of linearisability based on transforming sc histories by delaying returns to an associated flush event has also been explored [14, 10], allowing abstractions to remain atomic. Separately, working in the TSO memory model, Doherty and Derrick [12] study a weakening of linearisability using commutations allowed by the specification. In [9], motivated in part by hardware architectures such as Power and ARM, Derrick and Smith provide a framework for defining linearisability in relaxed memory models by allowing the observable order of the execution to be weaker (and hence different) from the full program order. The causal hb-linearisability definition of our paper follows the intuitions of [9]. In this paper, we identify the observable order in the context of AMT models. Since the AMT models provide a rich framework of relations to describe architectures, our methods apply to a wide class of architectures, thus including TSO, Power, and ARM. Our results also apply to the recent ARM8 proposal [24, 8] by identifying the “observed-before” order of that formalisation as the causal order.

Contextual refinement for the C11 memory model is studied by Batty et al. [4]. They consider histories of events constructed using *guarantees* and *deny* relations [11] — guarantees describe *happens-before* representing synchronisations internal to a library, whereas denies describe orders that cannot be enforced by a client due to the internal synchronisations within a library.

The use of happens-before in specifications to aid abstraction based reasoning has appeared in our prior paper [20]. We provided an order theoretic enhancement of linearisability that addresses TSO, PSO as well as JMM.

In this paper, we have been inspired by a simplified version of [4] (as specialised to handle the release-acquire atomics of C11) and the methods in our own prior paper [20]. In common with [20, 4, 10] and in contrast to [7], our definitions maintain the classical atomic and instantaneous view of method executions in linearisability. In common with all the above papers, we prove abstraction results. In common with [4], but in contrast with [7], we also prove composition results.

Acknowledgements. We thank our anonymous reviewers, whose comments have helped improve this paper.

References

1. Adve, S.V., Boehm, H.J.: Memory models: a case for rethinking parallel languages and hardware. Commun. ACM 53, 90–101 (2010)
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (1996)
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74 (2014)
4. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: POPL. pp. 235–248. ACM (2013)
5. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL. pp. 55–66. ACM (2011)
6. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI. pp. 68–78. ACM (2008)
7. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (ed.) ESOP. LNCS, vol. 7211, pp. 87–107. Springer (2012)
8. Deacon, W.: Arm64 cat file. `https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7` (2017)
9. Derrick, J., Smith, G.: An observational approach to defining linearizability on weak memory models. In: FORTE. pp. 108–123 (2017)
10. Derrick, J., Smith, G., Dongol, B.: Verifying linearizability on TSO architectures. In: IFM. LNCS, vol. 8739, pp. 341–356. Springer (2014)
11. Dodds, M., Feng, X., Parkinson, M.J., Vafeiadis, V.: Deny-guarantee reasoning. In: Castagna, G. (ed.) ESOP. LNCS, vol. 5502, pp. 363–377. Springer (2009)
12. Doherty, S., Derrick, J.: Linearizability and causality. In: SEFM. LNCS, vol. 9763, pp. 45–60. Springer (2016)
13. Dongol, B., Derrick, J.: Verifying linearisability: A comparative survey. ACM Comput. Surv. 48(2), 19 (2015)
14. Dongol, B., Derrick, J., Smith, G., Groves, L.: Defining correctness conditions for concurrent objects in multicore architectures. In: Boyland, J.T. (ed.) ECOOP. LIPIcs, vol. 37, pp. 470–494. Dagstuhl (2015)
15. Dongol, B., Groves, L.: Contextual trace refinement for concurrent objects: Safety and progress. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM. LNCS, vol. 10009, pp. 261–278 (2016)
16. Filipović, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52), 4379–4398 (2010)
17. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP(2). LNCS, vol. 6756, pp. 453–465 (2011)
18. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morg. Kauf. (2008)
19. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
20. Jagadeesan, R., Petri, G., Pitcher, C., Riely, J.: Quarantining weakness - compositional reasoning under relaxed memory models (extended abstract). In: Felleisen, M., Gardner, P. (eds.) ESOP. LNCS, vol. 7792, pp. 492–511. Springer (2013)
21. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Computers 46(7), 779–782 (1979)

22. Liang, H., Hoffmann, J., Feng, X., Shao, Z.: Characterizing progress properties of concurrent objects via contextual refinements. In: D'Argenio, P.R., Melgratti, H.C. (eds.) CONCUR. LNCS, vol. 8052, pp. 227–241. Springer (2013)
23. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL '05. pp. 378–391 (2005)
24. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. In: POPL (2018), to appear
25. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding power multiprocessors. In: PLDI. pp. 175–186. ACM (2011)
26. Sevcík, J.: Program Transformations in Weak Memory Models. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh (2008)
27. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
28. Shavit, N.: Data structures in the multicore age. Commun. ACM 54(3), 76–84 (2011)
29. Smith, G., Winter, K.: Relating trace refinement and linearizability. Formal Aspects of Computing pp. 1–16 (2017)

From Shapes to Amortized Complexity*

Tomáš Fiedor[1], Lukáš Holík[1], Adam Rogalewicz[1],
Moritz Sinn[3], Tomáš Vojnar[1], and Florian Zuleger[2]

[1] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
[2] TU Wien, Austria [3] St. Pölten University of Applied Sciences, Austria

Abstract. We propose a new method for the automated resource bound analysis of programs manipulating dynamic data structures built on top of an underlying shape and resource bound analysis. Our approach first constructs an integer abstraction for the input program using information gathered by a shape analyser; then a resource bound analyzer is run on the resulting integer program. The integer abstraction is based on shape norms — numerical measures on dynamic data structures (e.g., the length of a linked list). In comparison to related approaches, we consider a larger class of shape norms which we derive by a lightweight program analysis. The analysis identifies paths through the involved dynamic data structures, and filters the norms which are unlikely to be useful for the later bound analysis. We present a calculus for deriving the numeric changes of the shape norms, thereby generating the integer program. Our calculus encapsulates the minimal information which is required from the shape analysis.
We have implemented our approach on top of the Forester shape analyser and evaluated it on a number of programs manipulating various list and tree structures using the Loopus tool as the underlying bounds analyser. We report on programs with complex data structures and/or using complex algorithms that could not be analysed in a fully automated and precise way before.

1 Introduction

Automated resource bound analysis is an active field of research (for an overview we refer the reader e.g. to [1] and the references therein), which aims at developing tools and analysis techniques that allow developers to understand the performance of their code and to verify the resource consumption of their programs in case that bounding the resource consumption is a crucial correctness requirement.

The research of this paper is partly motivated by the experimental evaluation of our resource bound analysis tool Loopus [1], where we analysed a large number of C programs. Loopus computes resource bounds based on the updates to integer variables, however, has only a limited support for pointers. One of the results of our experiments was that missing pointer/shape analysis is the most frequent reason for the failure of Loopus to compute a resource bound. In [1] we report that we obtained bounds for 753 of the 1659 functions in our benchmark (45%), and that by a simple (but unsound) shape analysis we were able to increase the number of computed bounds to 1185 (71%).

In this paper, we study the automated resource bound analysis of heap-manipulating programs. We focus on the analysis of data structures as they can be found in systems code such as operating system kernels, compilers, or embedded systems. Performance is

* Supported by the Czech Science Foundation (project 17-12465S), the BUT FIT project FIT-S-17-4014, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602) and the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF).

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 205–225, 2018.
https://doi.org/10.1007/978-3-319-73721-8_10

a major concern in systems code and has led to the use of customised data structures and advanced data structures such as, e.g. red-black trees, priority heaps or lock-free linked lists. These data structures are complex and prone to introducing errors. Thus, automated tool support promises to increase the reliability of systems and can lead to a better user experience. Resource bound analysis of programs with data structures has been addressed only by a few publications [2,3,4,5,6]. In this paper we improve along several dimensions on these earlier results allowing the automated resource bounds analysis of heap-manipulating programs that cannot be handled by existing approaches.

Our approach. Our analysis works in three steps. We first run a shape analysis and annotate the program with the shape invariants. Then based on numeric information about the heap using the results from the shape analysis we create a corresponding integer abstraction of the program. Finally, we perform resource bound analysis purely on the integer program.

The numeric abstraction is based on *shape norms*, which are numerical measures on dynamic data structures (e.g. the length of a linked list). Our first contribution is the definition of a class of shape norms that express the longest distance between two points of interest in a shape graph and are defined in terms of basic concepts from graph theory. Our norms are parameterized by the program under analysis and are extracted in a pre-analysis (with a possibility of extending the initial set during the subsequent analysis); the extracted norms then correspond to selector paths found in the program.

Our second contribution is a calculus for our shape norms that allows to derive how the norms change along a program statement, i.e. if the norm is incremented resp. decremented or reset to some other expression. The calculus consists of two kinds of rules. (1) Rules that allow to directly infer the change of a norm and do not need to take additional information into account. (2) Rules that rely on the preceding shape analysis; the shape information is used there for (a) dealing with pointer aliasing and (b) deriving an upper bound on the value of a norm from the result of the shape analysis (if possible). We point out that the rules (2) encapsulate the points of the analysis where information about the shape is needed, and thus describe the minimal requirements on the preceding shape analysis. We believe that this separation of concern also allows the use of other shape analysers.

When creating the integer abstraction we could use all shape norms that we extracted from the program. However, we have an additional pre-analysis phase that eliminates norms that are not likely to be useful for the later bound analysis. This reduction of norms has the benefit that it keeps the number of variables in the integer abstraction small. The smaller number of additional variables increases the readability of the resulting integer abstraction and simplifies the developing and debugging of subsequent analyses. Additionally, the number of extracted norms can be quadratic in the size of the program; hence adding quadratically many variables can be prohibitively expensive and the pre-analysis is therefore crucial to the success of the later bound analysis.

Finally, we perform resource bound analysis on the created integer abstraction. This design decision has two advantages. First, we can leverage the existing research on bound analysis for integer programs and do not have to develop a new bound analysis. Second, being able to analyse not only shape but also integer changes has the advantage that we can analyse programs which mix integer iterations with data structure iterations; we illustrate this point by analysing the flagship example of [4], which combines iteration over data-structures and integer loops in an intricate way.

Implementation and Experiments. The generation of the integer program is implemented on top of the shape analyser Forester [7]. We use the Loopus tool [8,1] for inferring the computational complexity of the obtained integer abstractions. Our experimental evaluation demonstrates that the combination of these tools can yield a powerful analysis. We report on results for complex heap manipulating programs that could not be handled by previous approaches as witnessed by experimental evaluation against the tools AProVE [9] and COSTA [5]. We remark that our implementation leverages the strengths of both Forester and Loopus. We inherit the capabilities of Forester to analyse complex data structures, and report on analysis results for double-linked lists, trees, 2-level skip-lists, etc. Moreover, our analysis of shape norms is precise enough to leverage the capabilities of Loopus for amortized complexity analysis — we report on the amortized analysis of the flagship example of [4], whose correct linear bound has to the best of our knowledge never been inferred fully automatically.

Related work. The majority of the related approaches derive an integer program from an input heap-manipulating program followed by a dedicated analysis (e.g. termination or resource bounds) for arithmetic programs. The transformation has to be done conservatively, i.e. the derived integer program needs to simulate the original heap-manipulating program such that the results for the integer program hold for the original program. The related approaches differ in the considered numeric measures on the heap, the data structures that can be analysed and the degree of automation.

Several approaches have targeted restricted classes of data structures such as singly linked lists [10,11,12,13,14] or trees [15,16]. It is unclear how to generalise these results to composed or more complex data structures which require different numeric measures or combinations thereof.

A notable precursor to our work is the framework of [17] implemented in the THOR tool [18], which describes a general method for deriving integer abstractions of data structures. The automation of THOR, however, relies on the user for providing the shape predicates of interest (the implementation only comes with list predicates, further predicates have to be added by the user). Further, we found during initial investigations that THOR needlessly tracks shape sizes not required for a later termination or bounds analysis, which can quickly bloat the program under analysis.

A general abstract interpretation-style framework for combining shape and numerical abstract domains is described in [2]. The paper focuses on tracking of partition sizes, i.e. the only considered norm is the number of elements in a data structure. Our framework is orthogonal: we can express different norms, e.g. the height of a tree, which cannot be expressed in [2]; on the other hand, we use numeric information only in the second stage of the analysis which can be less precise than the reduced product construction of [2].

An automated approach to amortized complexity analysis of object-oriented heap-manipulating programs is discussed in [3]. The approach is based on the idea of associating a potential to (refinements of) data structure classes. Typing annotations allow to derive a constraint system which is then solved in order to obtain valid potential annotations. The implementation is currently limited to linear resource bounds and appears to be restricted to list-like data structures.

The idea of using potentials for the analysis of data structures is also investigated in [4]. The author extends separation logic with resource annotations exploiting the

idea of separation in order to associate resource units to every memory cell, resulting in an elegant Hoare-logic for resource analysis. The suggested approach is currently only semi-automated requiring the user to provide shape predicates and loop invariants manually.

In [5], the authors propose an automated resource analysis for Java programs, implemented in the COSTA tool. Their technique is based on abstracting arrays into their sizes and linked structures into the length of the longest chain of pointers terminated by `NULL`, followed by the construction and solving of a system of recurrence equations. However, cyclic lists and more complicated data structures such as DLLs, are, to the best of our knowledge, out of the capabilities of this technique as they require more general numeric size measures.

A recent paper investigates the automated resource analysis for Java programs and reports on its implementation in the AProVE tool [6], based on first translating a program to an integer transition system, and then using a bounds analyser to infer the complexity. The technique makes use of a single size-measure which is the number of nodes reachable from the heap node of interest together with the sum of all reachable integer cells. This norm is orthogonal to the norms considered in this paper. On the other hand, the norm of [6] does often not correspond to the size of interest: for example, in case of an iteration over the top-level list of a list of lists, the relevant norm is the length of the top-level list and not the number of total data structure elements; similarly in case of a search in a sorted tree: the relevant size measure is the height of a tree and not the number of elements. Moreover, it is unclear how the norm of [6] deals with cyclic data structures; while the number of reachable elements is well-defined, it is unclear if resp. how the norm changes when a pointer is advanced because the number of reachable nodes does not change.

Contributions. We summarise our contributions in this paper:

1. In comparison with related approaches we consider a larger class of shape norms.
2. We develop a calculus for deriving the numeric changes of the shape norms. The rules of our calculus precisely identify the information that is needed from a shape analyser. We believe that this definition of minimal shape information will be useful for the development of future resource bound analysis tools.
3. Our norms are not fixed in advance but mined from the program: We define a pre-analysis that reduces the number of considered norms. To our experience, this reduction is very useful during development of the resource analysis and for reporting the integer abstraction to the user.
4. We demonstrate in an experimental validation that we obtain a powerful analysis and report on complex data structure iterations that could not be analysed before.

2 A Brief Overview of the Proposed Approach

We are interested in deriving the computational complexity of Algorithm 1, where we understand the computational complexity as the total number of loop iterations. Our analysis infers an upper bound of the computational complexity by inferring a bound for the number of iterations of each loop and summing these loop bounds. The computation of other resource bounds can often be reduced to the computation of loop bounds in a similar way (we refer the reader to the discussion in [1] for more details).

```
1 x = list;
2 y = x;
3 while x ≠ NULL do
4 |   x = x.next;
5 |   if (*) then
6 |   |   while y ≠ x do
7 |   |   |   y = y.next;
```

Algorithm 1: A running example with computational complexity $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL} = \mathcal{O}(n)$.
* denotes nondeterministic choice.

```
1 x⟨next*⟩NULL = list⟨next*⟩NULL;
2 y⟨next*⟩x = 0;
3 while x⟨next*⟩NULL ≠ 0 do
4 |   x⟨next*⟩NULL−−;
  |   y⟨next*⟩x++;
5 |   if (*) then
6 |   |   while y⟨next*⟩x ≠ 0 do
7 |   |   |   y⟨next*⟩x−−;
```

Algorithm 2: A pure integer program corresponding to Algorithm 1

We thus limit the discussion in this paper to complexity bounds. A bound here is a symbolic expression in terms of the program variables. Our implementation computes complexity bounds with concrete constants—e.g., for Algorithm 1, we infer the bound $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL}$ where $list\langle\text{next}^*\rangle\texttt{NULL}$ is a shape norm (we discuss shape norms in more detail below). For ease of understanding and for comparison with related approaches, we also state asymptotic complexity bounds, which we obtain by replacing all shape norms with n, e.g., $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL} = \mathcal{O}(n)$.

We now present a brief overview of our approach on Algorithm 1, a simplified version of list partitioning. The outer loop at *line 3* iterates over the single linked list referenced by the variable `list`, at *line 5* the loop non-deterministically processes the partition of the list accumulated between variables `y` and `x`. We remark that deriving bounds for Algorithm 1 is challenging because (i) we have to reason not only about the lists x and y but also about the distance between these two pointers, and (ii) to infer the precise bound our reasoning must track the distance between x and y precisely rather than overapproximating it by the worst-case (which would lead to a quadratic bound). We sketch the main steps of our analysis below.

Shape analysis. The underlying shape analysis is run first. A successful run annotates each locations of the control-flow graph with a set of shape invariants and provides a guarantee that no safety violations, like e.g., a `NULL` pointer dereference, can occur during the program run. The shape invariants are needed to generate an integer abstraction of the program. Moreover, they can be leveraged to increase the precision of the subsequent bounds analysis, e.g., when the length of a path between variables y and x through the `next` selector is always constant. If the shape analysis fails, we end the analysis, as we will lack the necessary information to generate an integer program.

Deriving the candidate norms. We infer suitable candidate norms from the program control-flow as follows: (1) The loop header conditions define the set of candidate norms of the form $source\langle\text{re}^*\rangle destination$, where $source$ is a pointer variable, destination a distinct point (such as `NULL` or another pointer variable) and re is a placeholder for a regular expression over pointer selectors, which is filled in the next step. E.g., we can drive norm $x\langle\text{re}^*\rangle\texttt{NULL}$ from the condition $x \neq \texttt{NULL}$ in line 3 of Algorithm 1. We then derive the set of possible selector paths that may be traversed during the program run by a lightweight program analysis, in order to build the regular expression re: For our example we infer that at each iteration of the outer cycle variable `x` is moved by the selector `next`. We thus build the regular expression $next^*$ and obtain the complete candidate norm $x\langle\text{next}^*\rangle\texttt{NULL}$.

Analogically, we infer two candidate norms $x\langle\text{next}^*\rangle y$ and $y\langle\text{next}^*\rangle x$ for the inner loop at *line 6*. We thus obtain $\mathcal{N}_c = \{x\langle\text{next}^*\rangle\texttt{NULL}, x\langle\text{next}^*\rangle y, y\langle\text{next}^*\rangle x\}$ as the inital

set of candidate norms to be tracked. Note that this set is only an initial set and can be further extended during the generation of the integer program: E.g., when one of the norms $\mu_1 \in \mathcal{N}_c$ is reset to a norm $\mu_2 \notin \mathcal{N}_c$, μ_2 will be added to $\mathcal{N}_c$ and tracked.

Arithmetic program generation. For simplicity we consider only the norms $\mu_1 = x\langle\text{next}^*\rangle\texttt{NULL}$ and $\mu_2 = y\langle\text{next}^*\rangle x$ in our discussion, because these are sufficient to obtain a precise bound. We first translate the pointer conditions to corresponding integer conditions: The condition $x \neq \texttt{NULL}$ in line 3 is translated to the condition $\mu_1 \neq 0$ over norms. Analogically, the condition $x \neq y$ is translated to $\mu_2 \neq 0$.

We then derive norm updates (increases, decreases, resets) for each pointer instruction: *line 1* resets the norm μ_1 to $\mu_3 = list\langle\text{next}^*\rangle\texttt{NULL}$. We thus add μ_3 to the set of tracked norms $\mathcal{N}_c$. The execution of *line 4*, the instruction `x = x→next`, decrements norm μ_1 and increments μ_2. The instruction at *line 7* (in the inner loop) decrements the value of norm μ_2. By preserving the original control flow, but replacing all pointer instructions by the respective changes in norm values they provoke, we finally obtain the integer program depicted in Algorithm 2.

Bounds analysis. Finally, we apply the bounds analyser, Loopus [8], to infer a bound on the number of times that the loops at *line 3* and *line 6* of the integer abstraction Algorithm 2 can be iterated during the program run. In the following we comment on the analysis underlying Loopus (for a detailed description, we refer the reader to [1] or [19]): The norm $x\langle\text{next}^*\rangle\texttt{NULL}$ which decreases on the outer loop is initialized at *line 1* to the norm $list\langle\text{next}^*\rangle\texttt{NULL}$ and never reset. Hence the tool infers the bound $list\langle\text{next}^*\rangle\texttt{NULL}$ for the outer loop. The norm $y\langle\text{next}^*\rangle x$ which decreases in the inner loop of the integer program (*line 6*) is initialized to 0 at *line 2*, and never reset. However, at each execution of *line 4*, $y\langle\text{next}^*\rangle x$ is incremented by one (which models the execution of the statement $x = x \to next$ in the concrete program). Since the number of executions of *line 4* is bounded by the number of executions of the outer loop, the norm $y\langle\text{next}^*\rangle x$ is thus incremented at most $list\langle\text{next}^*\rangle\texttt{NULL}$ (the bound of the outer loop) times and hence the overall number of times the norm $y\langle\text{next}^*\rangle x$ may be decremented in the inner loop is bound by $list\langle\text{next}^*\rangle\texttt{NULL}$. The overall complexity of the example is the sum of both loop bounds, i.e. $2 \cdot list\langle\text{next}^*\rangle\texttt{NULL}$.

3 Preliminaries

This section introduces the basic notions used, the considered programs, as well as our requirements on the underlying shape analysis and on the way it should represent reachable memory configurations and their possible changes.

3.1 Program Model

For the rest of the paper, we will use $\mathbb{V}_p$ to denote the set of *pointer variables*, $\mathbb{V}_i$ the set of *integer variables*, $\mathbb{S}_p$ the set of *pointer selectors* (or fields) of dynamic data structures, and $\mathbb{S}_i$ the set of *integer selectors*. We assume all these sets to be finite and mutually disjoint. Let $\mathbb{V} = \mathbb{V}_p \cup \mathbb{V}_i$ be the set of all program variables and $\mathbb{S} = \mathbb{S}_p \cup \mathbb{S}_i$ be the set of all selectors. Finally, let NULL denote the *null* pointer and assume that $\texttt{NULL} \notin \mathbb{V} \cup \mathbb{S}$.

We consider *pointer manipulating program statements* from the set STMTS_p generated by the following grammar where $x, y \in \mathbb{V}_p$, $z \in \mathbb{V}_p \cup \{\texttt{NULL}\}$ and $sel \in \mathbb{S}_p$:

$$\texttt{stmt}_p ::= x = z \mid x = y \to sel \mid x \to sel = z \mid$$
$$x = \texttt{malloc()} \mid \texttt{free}(x) \mid x == z \mid x \neq z$$

Further, we consider *integer manipulating program statements* from the set STMTS_i generated by the following grammar where $x \in \mathbb{V}_i$, $y \in \mathbb{V}_p$, $sel \in \mathbb{S}_i$, $c \in \mathbb{Z}$, and f is an integer operation (more complex statements could easily be added too):

$$\texttt{stmt}_i ::= x = op \mid x = f(op, op) \mid y \to sel = op \mid x == op \mid x \neq op$$
$$op ::= c \mid x \mid y \to sel$$

Finally, we let $\text{STMTS} = \text{STMTS}_p \cup \text{STMTS}_i$.

Control-flow graphs. A *control-flow graph* (CFG) is a tuple $G = (\text{LOC}, T, l_b, l_e)$ where LOC is a finite set of program *locations*, $T \subseteq \text{LOC} \times \text{STMTS} \times \text{LOC}$ is a finite set of *transitions* (sometimes also called *edges*), $l_b \in \text{LOC}$ is the *initial (starting) location*, and $l_e \in \text{LOC}$ is the *final location*.

Let $G = (\text{LOC}, T, l_b, l_e)$ be a CFG. A path in G of length $n \geq 0$ is a sequence of transitions $t_0 \dots t_n = (l_0, st_0, l_1)(l_1, st_1, l_2) \dots (l_n, st_n, l_{n+1})$ such that $t_i \in T$ for all $0 \leq i \leq n$. We denote the set of all such paths by Φ_G. For a given location l, we denote by Φ_G^l the set of paths where $l_0 = l$. Given locations $l_1, l_2 \in \text{LOC}$, we say l_1 dominates l_2 (and denote it by $l_1 \succ l_2$) iff all paths to l_2 in $\Phi_G^{l_b}$ lead through l_1. We call a transition $(l, st, h) \in T$ a *back-edge* iff $h \succ l$. We call the location h a *loop header* and denote the set of its back-edges as T_h. Further, we denote the set of all loop headers as $\text{LH} \subseteq \text{LOC}$. Note that, for a loop header h_n of a loop nested in some outer loop with a loop header h_o, we have $h_o \succ h_n$.

Loops. Given a CFG $G = (\text{LOC}, T, l_b, l_e)$ with a set of loop headers LH, a loop L with a header $h_L \in \text{LH}$ is the sub-CFG $L' = (\text{LOC}', T_{|\text{LOC}'}, h_L, h_L)$ where $\text{LOC}' = \{l \in \text{LOC} \mid \exists n \geq 0\ \exists (l_0, st_0, l_1) \dots (l_n, st_n, l_{n+1}) \in \Phi_G : l_0 = l_{n+1} = h_L \wedge (\exists 0 \leq i \leq n : l = l_i) \wedge (\forall 1 \leq j \leq n : h_L \succ l_j)\}$, i.e., the set of locations on cyclic paths from h_L (but not crossing the header of any outer loop in which L might be nested), and $T_{|\text{LOC}'}$ is the restriction of T to LOC'. We will denote the set of all program loops as $\mathcal{L}$.

3.2 Memory Configurations

Let $\mathbb{V}_p$, $\mathbb{V}_i$, $\mathbb{S}_p$, and $\mathbb{S}_i$ be sets of pointer variables, integer variables, pointer selectors, and integer selectors, respectively, as defined in the previous. We view *memory configurations*, i.e., *shapes*, as triples $s = (M, \sigma, \nu)$ where (1) M is a finite set of memory locations, $\texttt{NULL} \notin M$, $M \cap \mathbb{Z} = \emptyset$, (2) $\sigma : (M \times \mathbb{S}_p \to M \cup \{\texttt{NULL}\}) \cup (M \times \mathbb{S}_i \to \mathbb{Z})$ is a function defining values of selectors, and (3) $\nu : (\mathbb{V}_p \to M \cup \{\texttt{NULL}\}) \cup (\mathbb{V}_i \to \mathbb{Z})$ is a function defining values of program variables. We denote the set of all such shapes by $\mathcal{S}$. Note that a shape is basically an oriented graph, also called a *shape graph*, with nodes from $M \cup \mathbb{Z} \cup \{\texttt{NULL}\}$, edges labelled by selectors, and some of the nodes referred to by the program variables. For simplicity, we do not explicitly deal with undefined values of pointers in what follows. For the purposes of our analysis, they can be considered equal to null values. If the program may crash due to using them, we assume this to be revealed by the shape analysis phase.

We assume that the shape analyser used within our approach works with a set $\mathcal{A}$ of *abstract shape representations* (ASRs), which can be automata, formulae, symbolic graphs, etc. This is, each ASR $A \in \mathcal{A}$ represents a (finite or infinite) set of shapes $[\![A]\!] \subseteq \mathcal{S}$. Allowing for disjunctive abstract representations, we assume that the shape analyser will label each location of the CFG of a program by a set of ASRs overapproximating the set of shapes reachable at that location. Moreover, we assume that the shape analyser introduces a special successor relation between ASRs whenever they label locations linked by a transition s.t. the statement of the transition may be executed between some shapes encoded by the ASRs. This leads to a notion of annotated CFGs defined below.

Annotated CFGs. An *annotated CFG* (ACFG) Γ is a triple $\Gamma = (G, \lambda, \rho)$ where $G = (\textsc{Loc}, T, l_b, l_e)$ is a CFG, $\lambda : \textsc{Loc} \to 2^{\mathcal{A}}$ is a function mapping locations to sets of ASRs generated by the underlying shape analyser for the particular locations, and $\rho \subseteq (\textsc{Loc} \times \mathcal{A}) \times (\textsc{Loc} \times \mathcal{A})$ is a *successor relation* on pairs of locations and ASRs where $((l_1, A_1), (l_2, A_2)) \in \rho$ iff $A_1 \in \lambda(l_1)$, $A_2 \in \lambda(l_2)$, and there is a transition $(l_1, st, l_2) \in T$ and shapes $s_1 \in [\![A_1]\!]$, $s_2 \in [\![A_2]\!]$ such that st transforms s_1 into s_2.

4 Numerical Measures on Dynamic Data Structures

Our approach uses a notion of *shape norms* based on regular expressions that encode sets of selector paths between some memory locations. Intuitively, we assume that the program needs to traverse these paths and hence their length determines (or at least contributes to) the complexity of the algorithm. Typically, one considers selector paths between two memory locations pointed by some pointer variables or between a location pointed by a variable and `NULL`. However, one can also use paths between a source location pointed by some variable and any location containing some specific data value.

For a concrete memory configuration, the numerical value of a shape norm corresponds to the *supremum* of the lengths of the paths represented by the regular expression of the norm. Indeed, in the worst case, the program may follow the longest (possibly cyclic and hence infinite) path in the memory. However, note that our analysis does usually not work with concrete values of shape norms since we work with ASRs and hence need to reason about the values of a given norm over potentially infinite sets of shapes. Instead, we track relative changes (i.e., increments, decrements) of the norms in a way consistent with all shapes in a given ASR. An exception to this is the case where the value of a norm is equal to a constant for all shapes in the ASR (e.g., after the statement $y = x \to next$, the distance from x to y via $next$ is always 1).

When analyzing a program, we first infer an initial set of *candidate norms* $\mathcal{N}_c$ (i.e., norms potentially useful for establishing resource bounds of the given program) from the CFG of the program—this set may later be extended if we realize some more norms may be useful. Subsequently, we derive as precisely as possible the effects (i.e., increments, decrements, or resets) that particular program statements have on the values of the candidate norms in shapes represented by the different ASRs obtained from shape analysis. The obtained set of shape norms $\mathcal{N}_c$ together with their relative changes could then be directly used to prove termination and to subsequently derive bounds on the loops by trying to form lexicographic norm vectors. The values of these vectors should be lexicographically ordered and have the property that the value is decreased by each loop iteration. However, we instead use $\mathcal{N}_c$ to generate a *numerical program* simulating the original program, which allows us to leverage the strength of current termination and

resource bounds analysers for numerical programs as well as to deal with termination and/or resource bounds arguments combining heap and numerical measures.

Below, we first formalize the notion of shape norms and then describe our approach to generating the initial set of candidate norms.

4.1 Shape Norms

Let $\mathbb{S}_p$ be the set of selectors. In what follows, we will use the set $\text{RE}_\mathbb{S}$ of restricted regular expressions $\texttt{re}$ over $\mathbb{S}_p$ defined as follows:

$$\texttt{re} ::= \texttt{ru}^* \qquad \texttt{ru} ::= sel \mid \texttt{ru} + \texttt{ru} \qquad sel \in \mathbb{S}.$$

Below, the $\texttt{ru}$ sub-expressions are called *regular units*, sometimes distinguishing *selector units* (sel) and *join units* ($\texttt{ru} + \texttt{ru}$). For $re \in \text{RE}_\mathbb{S}$, we denote the language of selector paths described by re as $\mathcal{L}_{re}$. Intuitively, when we analyse the control-flow graph of a program for traversals through selectors, a join unit corresponds to a branching of the control-flow, and the star expression ($\texttt{re}^*$) to a loop.

Our notion of selector regular expressions can be extended with *concatenation units* ($\texttt{ru}.\texttt{ru}$) and *nested star units* ($\texttt{ru}^*$), corresponding to sequences of unit traversals and nested loop traversals, respectively. Concatenation units are supported in our tool. However, since their introduction brings in many (quite technical) corner cases, we limit ourselves to the join units to simplify the presentation. On the other hand, extending the techniques below by nested stars seems to be more complicated, and we leave it for future work. Nevertheless, note that we did not find it much useful in our experiments as it would correspond to using the same variable as the iterator of several nested loops (while usually different pointer variables are used as the iterators of the loops).

Let $\mathbb{V}_p$ be a set of pointer variables and $\mathbb{S}_i$ a set of data selectors. We use $\mathcal{P} = \mathbb{V}_p \cup \{\texttt{NULL}\} \cup \{[.data = k] \mid k \in \mathbb{Z}, data \in \mathbb{S}_i\}$ to refer to locations of memory configurations (shapes) of a program. While $x \in \mathbb{V}_p$ denotes the location that is pointed by the pointer variable x, and $\texttt{NULL}$ denotes the special null location, $[.data = k]$ denotes any memory location whose selector $data$ has the value $k \in \mathbb{Z}$. A numerical measure μ on a memory configuration, i.e., a *shape norm*, is a triple $(x, \texttt{re}, y) \in \mathbb{V}_p \times \text{RE}_\mathbb{S} \times \mathcal{P}$. We will use $\mathcal{N}$ to denote the set of all shape norms, and, further, we will use $x\langle\texttt{ru}^*\rangle y$ as a shorthand for the triple $(x, \texttt{ru}^*, y) \in \mathcal{N}$.

As we have already mentioned above, we are interested in evaluating norms over ASRs, not over concrete shapes. Moreover, up to the cases where a norm has the same constant value for all shapes in an ASR, we are not interested in absolute values of the norms at all, and we instead track changes of the values of the norms only. However, in order to be able to soundly speak about such changes, we need to first define the value of a norm for a shape.

We will define the value of norms in terms of graphs. For this, we first define the notion of the height of a pointed graph. Then we describe how to obtain a pointed graph for a pair of a shape graph and a norm.

Pointed graphs. A *pointed graph* $\mathcal{G} = (N, E, n)$ consists of a set of nodes N, a set of directed edges $E \subseteq N \times N$ and, a source node $n \in N$. A path π is a finite sequence of nodes $n_0, \cdots, n_l$ such that $(n_i, n_{i+1}) \in E$ for all $0 \leq i < l$. We call $|\pi| = l$ the *length* of the path. We say π starts in n if $n_0 = n$. We define the *height* of $\mathcal{G}$ by setting $|\mathcal{G}| = \sup\{|\pi| \mid \text{path } \pi \text{ starts in } n\}$ where we set $\sup D = \infty$ for an infinite set $D \subseteq \mathbb{N}$. We note that, for a finite graph $\mathcal{G} = (N, E, n)$, we have $|\mathcal{G}| = \infty$ iff there is a cycle reachable from n.

Pointed graphs associated to shape graphs and null-terminated norms. We first consider norms μ that end in NULL, i.e, $\mu = x\langle \texttt{ru}^* \rangle \texttt{NULL}$. For a shape $s = (M, \sigma, \nu) \in \mathcal{S}$, we define the associated pointed graph $\mathcal{G}_s^{x\langle \texttt{ru}^* \rangle \texttt{NULL}} = (M \cup \{\texttt{NULL}\}, E, \nu(x))$ where $E = \{(n_1, n_2) \in (M \cup \{\texttt{NULL}\}) \times (M \cup \{\texttt{NULL}\}) \mid$ there is path from n_1 to n_2 in s s.t. the string of selectors along the path matches the regular expression $\texttt{ru}\}$.

Pointed graphs associated to shape graphs and non-null-terminated norms. We now consider a norm $\mu = x\langle \texttt{ru}^* \rangle y$ with $y \in \mathcal{P} \setminus \{\texttt{NULL}\}$. For a shape $s = (M, \sigma, \nu) \in \mathcal{S}$, we set $s(y) = \{\nu(y)\}$ for $y \in \mathbb{V}_p$, and $s(y) = \{m \in M \mid \sigma(m, data) = k\}$ for $y = [.data = k]$. We define the shape $s[y/\texttt{NULL}] = (M \setminus s(y), \sigma[y/\texttt{NULL}], \nu[y/\texttt{NULL}])$ where (1) $\sigma[y/\texttt{NULL}](m, sel) = \sigma(m, sel)$ if $m \notin s(y)$ and $\sigma[y/\texttt{NULL}](m, sel) = \texttt{NULL}$ otherwise, and (2) $\nu[y/\texttt{NULL}](x) = \nu(x)$ if $\nu(x) \notin s(y)$ and $\nu[y/\texttt{NULL}](x) = \texttt{NULL}$ otherwise. We define the associated pointed graph as $\mathcal{G}_s^{x\langle \texttt{ru}^* \rangle y} = \mathcal{G}_{s[y/\texttt{NULL}]}^{x\langle \texttt{ru}^* \rangle \texttt{NULL}}$.

Values of shape norms. We are now ready to define values of shape norms in shapes. In particular, the value of a norm $\mu \in \mathcal{N}$ in a shape $s \in \mathcal{S}$, denoted $\|\mu\|_s$, is a value from the set $\mathbb{N} \cup \{\infty\}$ defined such that $\|\mu\|_s = |\mathcal{G}_s^\mu|$. This is, the value of the norm μ in the shape s is defined as the height of the associated pointed graph.

The intuition behind the above definition is the following. The pointed graph associated to a norm $\mu = x\langle \texttt{ru}^* \rangle y$ makes the instances of the regular expression ru explicit. The height of the pointed graph corresponds to the longest chain of instances of the expression ru in the given shape graph. The intuition behind replacing the targets of norms with NULL stems from the fact that one either reaches the replaced target (and program will terminate naturally) or reaches the NULL, dereferences it and thus crashes (hence terminating unnaturally). However, since our method uses the results of a preceding shape analysis, we can assume memory safety and exclude termination by crash. In case there exists a cycle in the shape reachable from the source point x, the value of the norm is infinite. In such a case the norm is unusable for the later complexity analysis, hinting at the potential non-termination of the program under analysis.

We now generalize the notion of values of norms from particular shapes to ASRs. The value of a norm $\mu \in \mathcal{N}$ over a set of shapes given by an ASR $A \in \mathcal{A}$, denoted $\|\mu\|_A$, is a value from the set $\mathbb{N} \cup \{\infty, \omega\}$ defined such that $\|\mu\|_A = \sup_\omega\{\|\mu\|_s \mid s \in [\![A]\!]\}$ where (i) $\sup_\omega X = \omega$ iff $\infty \in X$ and (ii) $\sup_\omega(X) = \sup X$ otherwise. Intuitively, we need to distinguish the case when some of the represented shapes contains a cyclic selector path and the case where the ASR represents a set of shapes containing paths of finite but unbounded length (as, e.g., in the case when the ASR represents all acyclic lists of any length). Indeed, in the former case, the program may loop over the cyclic selector path while, in the latter case, it will terminate, but its running time cannot be bounded by a constant (it is bounded, e.g., by the length of the encountered list).

4.2 Deriving the Set of Candidate Shape Norms $\mathcal{N}_c$

We now discuss how we infer a suitable initial set of norm candidates. Note that this set is only an initial set of norm candidates that could be useful for inferring the bounds on the program loops. It is extended when tracking norm changes as discussed in Section 5. For each program loop L we derive a set of norm candidates in the following three steps:

1. We inspect all of the conditions of the loop L which involve pointer variables wrt the program model of Section 3.1 and declare each variable that appears in such

a condition as *relevant*. E.g., for our running example Algorithm 1 variables x and y are declared as relevant due to the condition $x \neq y$ in line *line 6*.

2. We iterate over all simple loop paths of L (a loop path is any path which stays inside the loop L, and starts from and returns to the loop header; a loop path is simple if it does not visit any location twice except for the loop header) and derive a set of selectors $S_x \subseteq \mathbb{S}_p$ for each relevant variable x: Given a simple loop path slp and a relevant variable x, we perform a symbolic backward execution to compute the effect of slp on x, i.e., we derive an assignment $x = exp$ such that exp captures how x is changed when executing slp. For example, for our running example Algorithm 1 we infer $x = x \to next$, $y = y$ for both simple loop paths of the outer loop and $y = y \to next$, $x = x$ for the single simple loop path of the inner loop. In case exp is of form $x \to sel$, i.e., the effect of the loop path is dereferencing variable x by some selector $sel \in \mathbb{S}_p$, we add sel to S_x. This basic approach can be easily extended to handle consecutive dereferences of the same pointer over different selectors: We can deal with expressions of the form $exp = x \to sel_1 \to sel_2$ by adding $sel_1.sel_2$ to S_x.

3. Finally, we consider all subsets $T \subseteq S_x$ and create norms for each $T = \{sel_1, ..., sel_l\}$ using the regular expression $join(T) = sel_1 + ... + sel_l$. The candidate norms $\mathcal{N}_L$ created for different forms of conditions of the loop L are given in the right column of Fig. 1. For example, for our running example in Sect. 2, we create norms $x\langle \text{next}^* \rangle \texttt{NULL}$ for the outer cycle, and norms $x\langle \text{next}^* \rangle y$ and $y\langle \text{next}^* \rangle x$ for the inner loop.

The overall set of tracked norm candidates $\mathcal{N}_c$ is set to the union of norm candidates over all loops in the program, i.e. $\mathcal{N}_c = \bigcup_{L \in \mathcal{L}} \mathcal{N}_L$. For each norm from $\mathcal{N}_c$ we track its size-changes, as we discuss in Section 5.

Condition of L	**Candidate Norms** $\mathcal{N}_L$
$x \circ y$	$\{\ x\langle join(\text{T})^* \rangle y \mid T \subseteq S_x \}$ $\cup \{\ y\langle join(\text{T})^* \rangle x \mid T \subseteq S_x \}$
$x \circ \texttt{NULL}$	$\{\ x\langle join(\text{T})^* \rangle \texttt{NULL} \mid T \subseteq S_x \}$
$x \to d \circ k$	$\{\ x\langle join(\text{T})^* \rangle [.data = k] \mid T \subseteq S_x \}$

Fig. 1: Norm candidates $\mathcal{N}_L$ for a loop L, $\circ \in \{=, \neq\}$

Note that we can optimize the size of $\mathcal{N}_c$ by pruning irrelevant norms, e.g. those that never decrease; the concrete heuristics are described in Section 6.2.

5 From Shapes to Norm Changes

In the previous section, we have shown how to derive an initial set of candidate norms $\mathcal{N}_c$ that are likely to be useful for deriving bounds on the number of executions of the different program loops. This section describes how to derive numerical changes of the values of these norms, allowing us to derive a numeric program simulating the original program from the point of view of its runtime complexity. During this process, new norms may be found as potentially useful, which leads to an extension of $\mathcal{N}_c$ and to a re-generation of the numeric program such that the newly added norms are also tracked.

In the numeric program, we introduce a *numeric variable* for each candidate norm. By a slight abuse of the notation, we use the norms themselves to denote the corresponding numeric variables, so, e.g., we will write $x\langle \text{u}^* \rangle \texttt{NULL} == 0$ to denote that the value of the numeric variable representing the norm $x\langle \text{u}^* \rangle \texttt{NULL}$ is zero. The values these variables may get are from the set $\mathbb{N} \cup \{\omega\}$ with omega representing an infinite distance (due to a loop in a shape). In what follows, we assume that any increment/decrement of ω yields ω again and that ω is larger than any natural number. Note that we do not need

a special value to represent ∞ for describing a finite distance without an explicit bound. For that, we will simply introduce a fresh variable constrained to be smaller than ω.

The *numeric program* is constructed using the ACFG $\Gamma = (G, \lambda, \rho)$ built on top of the CFG $G = (\text{LOC}, T, l_b, l_e)$ of the original program. The original control flow is preserved except that each location $l \in \text{LOC}$ is replaced by a separate copy for each ASR labelling it, i.e., it is replaced by locations (l, A) for each $A \in \lambda(l)$. Transitions between the new locations are obtained by copying the original transitions between those pairs of locations and ASRs that are related by the successor relation, i.e., a transition (l_1, st, l_2) is lifted to $((l_1, A_1), st, (l_2, A_2))$ whenever $((l_1, A_1), (l_2, A_2)) \in \rho$. Subsequently, each pointer-dependent condition labelling some edge in the extended CFG is translated to a condition on the numeric variables corresponding to the shape norms from $\mathcal{N}_c$. Likewise, each edge originally labeled by a pointer-manipulating statement is relabeled by numerical updates of the values of the concerned norm variables. Integer conditions and statements are left untouched.

Soundness of the abstraction. The translation of the pointer statements described below is done such that, for any path π in the CFG of a program and the shape s resulting from executing π, the values of the numeric norm variables obtained by executing the corresponding path in the numeric program conservatively *over-approximate* the values of the norms over s. This is, if the numeric variable corresponding to some norm μ can reach a value $n \in \mathbb{N} \cup \{\omega\}$ through the path π with pointer statements replaced as described below, then $||\mu||_s \leq n$. *As a consequence, we get that every bound obtained for the integer abstraction is a bound of the original program.*

Given the above, the translation of *pointer conditions* is easy. We translate each condition `x == NULL` to a disjunction of tests $x\langle \text{u}^* \rangle \texttt{NULL} == 0$ over all regular units u such that $x\langle \text{u}^* \rangle \texttt{NULL} \in \mathcal{N}_c$. Likewise, every condition `x == y` is translated to a disjunction of conditions of the form $x\langle \text{u}^* \rangle y == 0$ over all regular units u such that $x\langle \text{u}^* \rangle y \in \mathcal{N}_c$. Pointer inequalities are then translated to a negation of the conditions formed as above, leading to a conjunction of inequalities on numeric norm variables.

Handling *data-related pointer tests* of the form $\texttt{x} \to \texttt{data} == \texttt{y}$ is more complex. Consider such a test on an edge starting from a location-ASR pair (l, A). Currently, we can handle the test in a non-trivial way only if y evaluates to the same constant value in all shapes represented by A, i.e., if there is some $k \in \mathbb{N}$ such that $\nu(y) = k$ for all shapes $(M, \sigma, \nu) \in [\![A]\!]$. In this case, the test is translated to a disjunction of conditions of the form $x\langle \text{u}^* \rangle [.data = k] == 0$ over all regular units u such that $x\langle \text{u}^* \rangle [.data = k] \in \mathcal{N}_c$. Otherwise, the test is left out—a better solution is an interesting issue for future work, possibly requiring more advanced shape analysis and a tighter integration with it. Data-related pointer non-equalities can then again be treated by negation of the equality test (provided y evaluates to a constant value).

Finally, after a successful equality test (of any of the above kinds), all numeric norm variables that appeared in the disjunctive condition used are set to zero. All other variables (and all variables in general for an inequality test) keep their original value.

Next, we describe how we translate non-destructive, destructive, and data-related pointer statements other than tests. The translation can lead to decrements, resets, or increments of the numeric norm variables corresponding to the norms in $\mathcal{N}_c$. In case, we realize that we need some norm $\mu' \notin \mathcal{N}_c$ to describe the value of some current can-

didate norm $\mu \in \mathcal{N}_c$, we add μ' into $\mathcal{N}_c$ and restart the translation process (in practice, of course, the results of the previously performed translation steps can be reused). Such a situation can happen, e.g., when $\mathcal{N}_c = \{x\langle \text{next}^* \rangle \texttt{NULL}\}$ and we encounter an instruction `x = list`, which generates a reset of the norm $x\langle \text{next}^* \rangle \texttt{NULL}$ to the value of the norm $list\langle \text{next}^* \rangle \texttt{NULL}$. The latter norm is then added into $\mathcal{N}_c$.[1]

The rules for translating non-destructive, destructive, and data-related pointer updates to the corresponding updates on numeric norm variables are given in Fig. 2, 3, and 4, respectively. Before commenting on them in more detail, we first make several general notes. First, values of norms of the form $x\langle \text{u}^* \rangle x$ are always zero, and hence we do not consider them in the rules. Next, let $u = sel_1 + \ldots + sel_n$, $n \geq 1$, be a regular join unit. We will write $sel \in u$ iff $sel = sel_i$ for some $1 \leq i \leq n$. We denote new values of norms using an overline, and the old values without an overline. The norms that are not mentioned in a given rule keep implicitly the same value.

Finally, in rules describing how the value of a norm variable μ is changed by firing some statement between ASRs A_1 and A_2, we often use constructions of the form $\overline{\mu} \stackrel{\circ}{=} expr$ where $expr$ is an expression on norm variables. This construction constrains the new value of μ using the current values of norm variables or using directly the ASRs encountered, depending on what of this is more precise. First, if μ has the same natural value in all shapes in $[\![A_2]\!]$, i.e., if $\|\mu\|_{A_2} \in \mathbb{N}$, then we let $\overline{\mu} = \|\mu\|_{A_2}$. Otherwise, if the value of μ is infinite in A_1 and unbounded but finite in A_2, i.e., if $\|\mu\|_{A_1} = \omega$ and $\|\mu\|_{A_2} = \infty$, we constrain the new value of μ by the constraint $\overline{\mu} = v \wedge v < \infty$ where v is a fresh numeric variable.[2] The same constraint with a fresh variable is used when $\|\mu\|_{A_2} = \infty$ and $expr = \omega$. Otherwise, we let $\overline{\mu} = expr$.

The described translation allows for sound resource bounds analysis. Indeed, for each run of the original pointer program, there will exist one run in the derived numeric program where the norms get exact/overapproximated values. Provided that the underlying bounds analyser is sound in that it returns worst case bounds, the bounds obtained for the numeric program will not be smaller than the bounds of the original program.

5.1 Non-Destructive Pointer Updates

We now comment more on the less obvious parts of the rules for non-destructive pointer updates from Fig. 2. Concerning the rule for $x = \texttt{NULL}$, Case 1 reflects the fact that we always consider all paths from x limited by either the designated target w or, implicitly, `NULL`. Hence, after $x = \texttt{NULL}$, the distance is always 0. Likewise, in Case 1 of $x = malloc()$, the distance is always 1 as we assume all fields of the newly allocated cell to be nullified, and so the paths consist of the newly allocated cell only. Case 2 of $x = malloc()$ is based on that we assume the newly allocated cell to be unreachable

[1] Alternatively, one could use a more complex initial static analysis that would cover, although may be less precisely, even such dependencies among norms.

[2] Intuitively, this case is used, e.g., when $\mu = x\langle \text{n}^* \rangle \texttt{NULL}$, and the encountered pointer statement cuts an ASR representing cyclic lists of any length pointed by x to an ASR representing acyclic `NULL`-terminated lists pointed by x. Naturally, when one subsequently starts a traversal of the list, it will terminate though in an unknown number of steps.

[x = NULL]

$$\frac{\forall w \in \mathcal{P},\ \forall z \in \mathbb{V}_p \setminus \{x\}}{\begin{array}{rcll}\overline{x\langle \mathrm{u}^* \rangle w} & = & 0 & (1)\\ \overline{z\langle \mathrm{u}^* \rangle x} & \circeq & z\langle \mathrm{u}^* \rangle \mathtt{NULL} & (2)\end{array}}$$

[x = malloc()]

$$\frac{\forall w \in \mathcal{P} \setminus \{x\},\ \forall z \in \mathbb{V}_p}{\begin{array}{rcll}\overline{x\langle \mathrm{u}^* \rangle w} & = & 1 & (1)\\ \overline{z\langle \mathrm{u}^* \rangle x} & \circeq & z\langle \mathrm{u}^* \rangle \mathtt{NULL} & (2)\end{array}}$$

[free(x)]

$$\frac{\forall z \in \mathbb{V}_p,\ \forall w \in \mathcal{P}}{\overline{z\langle \mathrm{u}^* \rangle w} \circeq \begin{cases} z\langle \mathrm{u}^* \rangle x & \mathtt{AllPathsThr}(A_1, u, z, w, x)) \\ z\langle \mathrm{u}^* \rangle w & \text{otherwise} \end{cases}}$$

[x = y→n (alias)]

$$\frac{\exists v \in \mathtt{AliasNext}(A_1, y, \mathtt{n}) \quad \forall w \in \mathcal{P}\ \forall z \in \mathbb{V}_p}{\begin{array}{rcll}\overline{x\langle \mathrm{u}^* \rangle w} & \circeq & v\langle \mathrm{u}^* \rangle w & (1)\\ \overline{z\langle \mathrm{u}^* \rangle x} & \circeq & z\langle \mathrm{u}^* \rangle v & (2)\end{array}}$$

[x = y→n (non-unit)]

$$\frac{\mathtt{n} \notin u,\ \forall w \in \mathcal{P},\ \forall z \in \mathbb{V}_p}{\begin{array}{rcll}\overline{x\langle \mathrm{u}^* \rangle w} & \circeq & \omega & (1)\\ \overline{z\langle \mathrm{u}^* \rangle x} & \circeq & \omega & (2)\end{array}}$$

[x = y→n (unit)]

$$\frac{\begin{array}{c}\mathtt{n} \in u,\ x \neq y,\ \forall t \in \mathtt{Alias}(A_1, y)\\ \forall s \in \mathtt{MayAlias}(A_1, y)\\ \forall w \in \mathcal{P} \setminus \mathtt{MayAlias}(A_1, y)\\ \forall z \in \mathbb{V}_p \setminus \mathtt{Alias}(A_1, y)\end{array}}{\begin{array}{rcll}\overline{t\langle \mathrm{u}^* \rangle x} & \circeq & t\langle \mathrm{u}^* \rangle \mathtt{NULL} & (1)\\ \overline{x\langle \mathrm{u}^* \rangle s} & \circeq & y\langle \mathrm{u}^* \rangle \mathtt{NULL} - 1 & (2)\\ \overline{x\langle \mathrm{u}^* \rangle w} & \circeq & y\langle \mathrm{u}^* \rangle w - 1 & (3)\\ \overline{z\langle \mathrm{u}^* \rangle x} & \circeq & z\langle \mathrm{u}^* \rangle y + \overline{y\langle \mathrm{u}^* \rangle x} & (4)\end{array}}$$

[x = y]

$$\frac{\forall z \in \mathbb{V}_p\ \forall w \in \mathcal{P}}{\overline{x\langle \mathrm{u}^* \rangle w} \circeq y\langle \mathrm{u}^* \rangle w\ (1) \quad \overline{z\langle \mathrm{u}^* \rangle x} \circeq z\langle \mathrm{u}^* \rangle y\ (2)}$$

Fig. 2: Translation rules for *non-destructive pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a non-destructive pointer update with $x, y \in \mathbb{V}_p$, $n \in \mathbb{S}_p$. For all rules with left-hand side of form $\overline{a\langle \mathrm{u}^* \rangle b}$, u ranges over all regular units such that $a\langle \mathrm{u}^* \rangle b \in \mathcal{N}_c$. If the norms used on the right-hand side of any of the applied rules is not in $\mathcal{N}_c$, it is added into $\mathcal{N}_c$, and the analysis is re-run with the new $\mathcal{N}_c$.

from other memory locations, and so any path taken from another memory location towards x will implicitly be bounded by NULL.[3]

Concerning the rules for $free(x)$, the predicate $\mathtt{AllPathsThr}(A, u, z, w, x)$ holds iff all paths over selector sequences matching u^* between the location z and the location w go through x in all shapes in $[\![A]\!]$. In this case, clearly, all paths from z to w are shrunk to paths to x by $free(x)$ as x becomes undefined (which we take as equal to NULL for our purposes). Otherwise, we take the old value of the norm since it either stays the same or perhaps gets shorter but in some shapes only.

Concerning the rules for $x = y \to n$, we first note that, if applicable, the "alias" rule has priority. It is applied when the n-successor of y is pointed by some variable v in all shapes in $[\![A_1]\!]$. Formally, $v \in \mathtt{AliasNext}(A, y, \mathtt{n})$ iff $\forall (M, \sigma, \nu) \in [\![A]\!] : \sigma(\nu(y), n) = \nu(v)$. Such an alias can be used to define norms based on x by copying those based on v. Of course, the distance from x to v after the update should be zero, which is assured by the $\circeq$ operator. If there is no such v, and n does not match u, we can only limit the new value of the norm based on the ASR, which is again taken care by the $\circeq$ operator (otherwise we take the worst possibility, i.e., ω).

The most complex rule is that for $x = y \to n$ when there is no alias for the n-successor of y, and n matches u. First, note that the rule is provided for the case of x being a different variable than y only. We assume statements $x = x \to n$ to be transformed to a sequence $y = x; x = y \to n$; for a fresh pointer variable y. In the rules, we then use the following must- and may-alias sets: $\mathtt{Alias}(A, y) = \{v \in \mathbb{V}_p \mid \forall (M, \sigma, \nu) \in [\![A]\!] : \nu(y) = \nu(v)\}$ and $\mathtt{MayAlias}(A, y) = \{v \in \mathbb{V}_p \mid \exists (M, \sigma, \nu) \in [\![A]\!] : \nu(y) = \nu(v)\}$.

Concerning Case 1, note that u can be a join unit and u^* can match several paths from y that need not go to the new position of x at all (and hence can stop only when reaching NULL), or they can go there, but as there is no variable pointing already to the

[3] We assume that the preceding shape analysis will discover potential problems with a location being freed and re-allocated with some dangling pointers still pointing to it (the ABA problem).

$$\frac{\begin{array}{c}\textbf{[x}\rightarrow\textbf{n = }\texttt{NULL}\textbf{ (unit)]}\\ \texttt{n} \in u,\ \forall z \in \mathbb{V}_p,\ \forall w \in \mathcal{P}\end{array}}{\overline{z\langle \text{u}^* \rangle w} \mathrel{\mathring{=}} \begin{cases} z\langle \text{u}^* \rangle x + 1 & \texttt{AllPathsThrFld}(A_1, u, z, w, x, n) \\ z\langle \text{u}^* \rangle w & \text{otherwise} \end{cases}}$$

$$\frac{\begin{array}{c}\textbf{[x}\rightarrow\textbf{n = y (unit)]}\\ \texttt{n} \in u,\ \forall s_1 \in \texttt{Alias}(A_1, x),\ \forall s_2 \in \texttt{MayAlias}(A_1, x)\\ \forall t_1 \in \texttt{Alias}(A_1, y),\ \forall t_2 \in \texttt{MayAlias}(A_1, y)\\ \forall w \in \mathcal{P} \setminus (\texttt{Alias}(A_1, x) \cup \texttt{Alias}(A_1, y))\\ \forall z \in \mathbb{V}_p \setminus (\texttt{MayAlias}(A_1, x) \cup \texttt{MayAlias}(A_1, y))\end{array}}{\begin{array}{rll}\overline{s_1\langle \text{u}^* \rangle t_1} \mathrel{\mathring{=}} & s_1\langle \text{u}^* \rangle t_1 & (1)\\ \overline{s_2\langle \text{u}^* \rangle w} \mathrel{\mathring{=}} & \begin{cases} \omega & \texttt{BadLoopClosed}(A_2, u, y, x, w) \\ \overline{s_2\langle \text{u}^* \rangle y} + y\langle \text{u}^* \rangle w & \text{otherwise} \end{cases} & (2)\\ \overline{t_2\langle \text{u}^* \rangle w} \mathrel{\mathring{=}} & \begin{cases} \omega & \texttt{BadLoopClosed}(A_2, u, y, x, w) \\ t_2\langle \text{u}^* \rangle w & \text{otherwise} \end{cases} & (3)\\ \overline{z\langle \text{u}^* \rangle w} \mathrel{\mathring{=}} & \begin{cases} z\langle \text{u}^* \rangle x + \overline{x\langle \text{u}^* \rangle w} & \texttt{AllPathsThr}(A_2, u, z, w, x)) \\ \max(z\langle \text{u}^* \rangle x + \overline{x\langle \text{u}^* \rangle w}, z\langle \text{u}^* \rangle w) & \texttt{SomePathsThr}(A_2, u, z, w, x)) \\ z\langle \text{u}^* \rangle w & \text{otherwise} \end{cases} & (4)\end{array}}$$

Fig. 3: Translation rules for *destructive pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a destructive pointer update with $x, y \in \mathbb{V}_p$, $n \in \mathbb{S}_p$. The treatment of the regular units u is the same as in Fig. 2.

new position of x, we anyway have to approximate such paths by extending them up to NULL. The case when the only path to x is via n will then be solved by the $\mathring{=}$ operator. The must aliases of y can naturally be treated in an equal way as y in the above.

In Case 2, start by considering paths from x to y. Since we have no alias of the n-successor of y that could help us define the value of the norm, we have to approximate the distance from x to y by extending the paths from x up until NULL. Further note that such paths are a subset of those from y to NULL (since the new position of x is a successor of y). We can thus use $y\langle\text{u}^*\rangle\texttt{NULL}$ to approximate $\overline{x\langle\text{u}^*\rangle\texttt{NULL}}$. However, we can constrain the latter distance to be smaller by one. Indeed, if the longest path from y to NULL does not go through the new position of x, the distance from x to NULL is at least by one smaller. On the other hand, if the longest path goes through the new position of x, then we save the step from y to the new position of x. The same reasoning then applies for any variable that may alias y—for those that cannot alias it, one can do better as expressed in the next case.

In Case 3, one can use a similar reasoning as in Case 2 as the paths from y to w include those from the new position of x to w. Note, however, that this reasoning cannot be applied when y may alias with w. In such a case, their distance may be zero, and the distance from the new position of x to w can be bigger, not smaller. Finally, to see correctness of Case 4, note that should there be a longer path over u from z to the n-successor of y than going through y, this longer path will be included into the value of the norm for getting from z to y too since the norm takes into account all u paths either going to y or missing it and then going up until NULL (or looping).

The intuition behind the rule for $x = y$ is similar to the other statements.

5.2 Destructive Pointer Updates

We now proceed to the rules for destructive pointer statements shown in Fig. 3. We start with the translation for the statement $x \to n = \texttt{NULL}$, considering the case of n being a unit, i.e., $n \in u$. After this statement, the distance from any source memory location z to any target memory location w either stays the same or decreases. The latter happens

$$\frac{\exists k \in \mathbb{Z} : \texttt{ValIsConst}(A_1, y, k),\ \forall z \in \mathbb{V}_p \setminus \{x\},\ \forall l \in \mathbb{Z} \setminus \{k\}}{\begin{array}{lll} \overline{x\langle \mathtt{u}^* \rangle [.d = k]} = 0 & & (1) \\ \overline{x\langle \mathtt{u}^* \rangle [.d = l]} \mathrel{\mathring{=}} \begin{cases} x\langle \mathtt{u}^* \rangle \mathtt{NULL} & \texttt{ValMayBe}(A_1, x, d, l) \\ x\langle \mathtt{u}^* \rangle [.d = l] & \text{otherwise} \end{cases} & & (2) \\ \overline{z\langle \mathtt{u}^* \rangle [.d = k]} \mathrel{\mathring{=}} \begin{cases} z\langle \mathtt{u}^* \rangle x & \texttt{AllPathsThr}(A_1, u, z, [.d = k], x) \\ z\langle \mathtt{u}^* \rangle [.d = k] & \text{otherwise} \end{cases} & & (3) \\ \overline{z\langle \mathtt{u}^* \rangle [.d = l]} \mathrel{\mathring{=}} \begin{cases} z\langle \mathtt{u}^* \rangle \mathtt{NULL} & \texttt{ValMayBe}(A_1, x, d, l) \\ z\langle \mathtt{u}^* \rangle [.d = l] & \text{otherwise} \end{cases} & & (4) \end{array}}\ [\mathtt{x{\to}d = y\ (data\text{-}const)}]$$

$$\frac{(\neg \exists k \in \mathbb{Z} : \texttt{ValIsConst}(A_1, y, k)),\ \forall z \in \mathbb{V}_p,\ \forall l \in \mathbb{Z}}{\overline{z\langle \mathtt{u}^* \rangle [.d = l]} \mathrel{\mathring{=}} \begin{cases} z\langle \mathtt{u}^* \rangle [.d = l] & z\langle \mathtt{u}^* \rangle [.d = l] < z\langle \mathtt{u}^* \rangle x \\ z\langle \mathtt{u}^* \rangle [.d = l] & \neg\texttt{ValMayBe}(A_1, x, d, l) \\ z\langle \mathtt{u}^* \rangle \mathtt{NULL} & \end{cases}}\ [\mathtt{x{\to}d = y\ (data\text{-}unknown)}]$$

Fig. 4: Translation rules for *data-related pointer updates*. The rules are assumed to be applied between location-ASR pairs (l_1, A_1) and (l_2, A_2) linked by an edge labelled by a data-related pointer update with $x \in \mathbb{V}_p$, $y \in \mathbb{V}_i$, $d \in \mathbb{S}_i$. The treatment of the regular units u is the same as in Fig. 2.

when the changed n-selector of x influences the longest previously existing path from z to w. Identifying this case in general is difficult, but one can reasonably recognise it in common ASRs at least in the situation when all paths between z and w whose selector sequences match u^* go through the n-selector of the memory location marked by x in all shapes represented by the ASR A_1, i.e., $[\![A_1]\!]$. We denote this fact by the predicate $\texttt{AllPathsThrFld}(A_1, u, z, w, x, n)$. In this case, the new distance between z and w clearly corresponds to the old distance between z and x plus one (for the step from x to NULL). In all other cases, we conservatively keep the old value of the distance (up to it can be reduced by the $\mathring{=}$ operator as usual).

Concerning the statement $x \to n = y$, the distance between x and y (and their aliases) can stay the same or get shortened. In Case 1 of the rule for this statement, the latter is reflected in the use of the $\mathring{=}$ operator. In Case 2, we use the predicate $\texttt{BadLoopClosed}(A_2, u, y, x, w)$ to denote a situation when the statement $x \to n = y$ closes a loop (over the u selectors) in at least some shape represented by A_2 such that w does not appear in between of y and x in the loop. Naturally, in such a case, the distance between x (or any of its may-aliases) and w is set to ω. Note that the may-alias is needed in this case since it is enough that this problematic situation arises even in one of the concerned shapes. As for correctness of the other variant of Case 2, note that if there are paths over u^* from x to w not passing through y, they will be covered by $\overline{x\langle \mathtt{u}^* \rangle y}$, which will consider such paths extended up until NULL. In Case 3, note that if the loop is not closed, then the paths from y to w are not influenced.

In Case 4, if all paths from z to w in the shapes represented by A_2 go through x, we can take the original distance of z and x, which does not change between A_1 and A_2 as the change happens after x, and then add the new distance from x to w. If no path from z to w passes x, the distance is not influenced by the statement. If some but not all of the paths pass x, we have to take the maximum of the two previous cases.

As for non-unit cases of the above two statements, i.e., the case when $n \notin u$, the norms do not change since the paths over u^* do not pass the changed selector.

5.3 Data-Related Pointer Updates

Our rules for translating data-related pointer updates are given in Fig. 4. The first of them applies in case the value being written into the data field d of the memory location

pointed by x is constant over all shapes represented by the ASR A_1, i.e., if there is some constant $k \in \mathbb{Z}$ such that $\forall(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = k$. This fact is expressed by the ValIsConst(A_1, y, k) predicate. In this case, after the statement $x \to d = y$, the distance from x to a data value k becomes clearly zero. Case 2 captures the fact that if the d-field of x may be l in at least one shape represented by A_1, i.e., if $\exists(M, \sigma, \nu) \in [\![A]\!] : \nu(y) = l$, which is expressed by the ValMayBe(A_1, x, d, l) predicate, the new distance of x to a data value l is approximated by its distance to NULL. The reason is that the old data value is re-written, and one cannot say whether another data field with the value l may be reached before one gets to NULL. Otherwise, the norm keeps its original value. Case 3 covers the distance from a location z other than x to a data value k. This distance clearly stays the same or can get shorter after the statement. We are able to safely detect the second scenario when all paths from z to a data value k lead through x. In that case, the distance from z to a data value k shrinks to that from z to x. Otherwise, we conservatively keep the norm value unchanged. Finally, Case 4 is an analogy of Case 2.

In case the value being written through a data selector is not constant, which is covered by the second rule of Fig. 4, our approach is currently rather conservative. We keep the original value of the norms between z and a data value l if either this data value is always reached from z before x is reached (the norm takes into account the first occurrence of the data value) or if the re-written value of the data field d of x is not l in any of the shapes represented by A_1 (and hence the original value of the norm is not based on the distance to this particular field). In such a case, the distance between z and the data value l does surely not change. Otherwise, we conservatively approximate the new distance between z and the data value l by the distance over paths matching u^* from z up until NULL.

The stress on handling constant values of data may seem quite restricted, but it may still allow one to verify a lot of interesting programs. The reason is that often the programs use various important constants (like 0) to steer their control flow. Moreover, due to data-independence, it is often enough to let programs work with just a few constant values—c.f., e.g., [20,21,22] where just a few data values ("colors") are used when checking various advanced properties of dynamic data structures. Still a better support of data is an interesting issue for future work.

6 Implementation and Experiments

We have implemented our method in a prototype tool called RANGER. The implementation is based on the *Forester* shape analyser [23,22], which represents sets of memory shapes using so-called *forest automata* (FAs). As a back-end *bounds analyser* for the generated numeric programs, we use the *Loopus* tool [8]. We evaluated RANGER on a set of benchmarks including programs manipulating various complex data structures and requiring amortized reasoning for inferring precise bounds. The experimental results we obtained are quite encouraging and show that we were able to leverage both the precise shape analysis of complex data structure provided by Forester as well as the amortized analysis of loop bounds provided by Loopus and for the first time fully-automatically and precisely analyse some challenging programs.

In the rest of the section, we first briefly introduce the Forester tool in some more detail and discuss how we implemented our approach on top of it. Next, we mention various further optimizations we included into the implementation. Then, we present the experiments we performed and their results.

6.1 Implementation on Top of Forester

The Forester shape analyser represents particular shapes by decomposing them into *tuples* of *tree components*, and hence *forests*. In particular, each memory location that is `NULL`, pointed by a pointer variable, or that has multiple incoming pointers becomes a so-called *cut-point*. Shape graphs are cut into tree components at the cut-points, and each cut-point becomes the root of one of the tree components. Leaves of the tree components may then refer back to the roots, which can be used to represent both loops in the shapes as well as multiple paths leading to the same location. Of course, Forester does not work with particular shapes but with sets of shapes. This leads to a need of dealing with tuples of sets of tree components, which are finitely represented using finite *tree automata* (TAs). A tuple of TAs then forms an FA, which we use as the ASR in our implementation.

Hence, we need to be able to implement all the operations used on ASRs in the previous section on FAs. Fortunately, it turns out that this is not at all difficult.[4] In particular, we can implement the various operations by searching through the particular TAs of an FA, following the TA transitions that match the relevant unit expressions.[5] We can then, e.g., easily see whether the distance between some memory locations is constant, finite but unbounded, or infinite. It is constant if the given memory locations are linked by paths in the structure of the involved automata that are of the given constant length. It is finite but unbounded if there is a loop in the TA structure in between the concerned locations (allowing the TA to accept a sequence of any finite length). Finally, the distance is infinite if some path from the source location leads—while not passing through the target location—to some of the roots, which is then in turn referenced back from some leaf node reachable from it. Likewise, one can easily implement checks whether all paths go (or at least some path goes) through some location, whether some variables are aliased (in Forester, this simply corresponds to the variables being associated with the same root), or whether a loop is closed by some destructive update (which must create a reference from a leaf back to a loop).

6.2 Optimizations of the Basic Approach

In RANGER, we use several heuristic optimizations to reduce the size of the generated numeric program. First, we do not translate each pointer statement in isolation as described in Section 5. Instead, we perform the translation per *basic blocks*. Basically, we take the blocks written in the static single assignment form, translate the statements in the blocks as described in Section 5, and then perform various standard simplifications

[4] Based on our experience with other representations of sets of shapes, such as separation logic or symbolic memory graphs, we believe that it would not be difficult with other shape representations either.

[5] In RANGER, we support even concatenation units to some degree, which requires us to look at sequences of TA transitions to match a single unit.

of the generated numeric constraint (evaluation of constant expressions, copy propagation, elimination of variables) using the SMT solver Z3 [24]. In our experience, the size of the generated numeric program can be significantly reduced this way.

Our second optimization aims at reducing the *number of tracked norms*. For that, we use a simple heuristic exploiting the underlying shape analysis and the principle of *variable seeding* [14]. Basically, for each pointer variable x used as a source/target of some norm in $\mathcal{N}_c$, we create a shadow variable x', and remember the position of x at the beginning of a loop by injecting a statement $x' = x$ before the loop. We then use our shape analyser on the extended code to see whether the given variable indeed moves towards the appropriate target location when the loop body is fired once. If we can clearly see that this is not the case due to, e.g., the variable stays at the same location, we remove it from $\mathcal{N}_c$. For illustration, in our example from Section 2, we generate two norms $y\langle \text{next}^* \rangle x$ and $x\langle \text{next}^* \rangle y$ for the loop at *line 6*. Using the above approach, we can see that x is never moved, $x\langle \text{next}^* \rangle y$ is never decreased, and so we can discard it. Moreover, we check which norms decrease at which loop branches (or, more precisely, that cannot be excluded to decrease) and prune away norms that decrease only when some other norm is decreased—we say that such a norm is *subsumed*.

Finally, we reduce the size of the resulting numeric program by taking into account only those changes (resets, increments, and decrements) of the norms whose effect can reach the loop for whose analysis the norm is relevant. For that, we use a slight adaptation of the reset graphs introduced in [1].

6.3 Experimental Evaluation

Our experiments were performed on a machine with an Intel Core i7-2600@3.4 GHz processor and 32 GiB RAM running Debian GNU/Linux. We compared our prototype RANGER with two other tools: APROVE and COSTA. These two tools are, to the best of our knowledge, the closest to RANGER and represent the most recent advancements in bounds analysis of heap-manipulating programs. However, note that both of the tools work over the Java bytecode, and thus we had to translate our benchmarks to Java. For our tool, we report three times — the running times of the shape analysis of Forester (**SA**), generation of the integer program (**IG**), and bounds analysis in Loopus (**BA**). For the other tools, we report times as reported by their web interface[6].

Further, from the outputs of the tools, we extracted the reported complexity of the main program loop, and, if needed, simplified the bounds to the big $\mathcal{O}$ notation. We remark that COSTA uses path-based norms (i.e. a subset of our norms), so it is directly comparable with RANGER. APROVE, however, uses norms based on counting all reachable elements, and is therefore orthogonal to us. But, their norm is always bigger than our norms, thus if it reports an equal or bigger computational complexity we can meaningfully compare the results.

The results are summarized in Table 1. We use TIMEOUT(60S) if a time-out of 60 seconds was hit, ERROR if the tool failed to run the example[7], and UNKNOWN if the tool could not bound the main loop of the example. We divided our benchmarks to three distinct categories. The BASIC category consists of simple list structures — Single-Linked Lists (SLL), Circular Single-Linked Lists (CSLL). In the ADVANCED

[6] We could not directly compare the tools on the same machine due to the tool availability issues.

[7] However, we verified that all our examples are syntactically correct.

Table 1: Experimental results.

Benchmark	Short description	Real bounds	RANGER Bound	RANGER SA	RANGER IG	RANGER BA	APROVE Bound	APROVE Time(web)	COSTA Bound	COSTA Time(web)
BASIC										
SLL-CST	Constant-length SLL Traversal	$\mathcal{O}(1)$	$\mathcal{O}(1)$	0.002s	0.023s	0.011s	$\mathcal{O}(1)$	3.664s	$\mathcal{O}(n)$	0.251s
SLL	SLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.012s	0.087s	0.040s	$\mathcal{O}(n)$	6.434s	$\mathcal{O}(n)$	0.441s
SLL-NESTED	SLL with non-reset nested traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.027s	0.256s	0.057s	$\mathcal{O}(n)$	6.361s	$\mathcal{O}(n^2)$	1.582s
SLL-INT	SLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.037s	0.275s	0.057s	$\mathcal{O}(n)$	8.945s	$\mathcal{O}(n)$	0.921s
CSLL	CSLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.013s	0.086s	0.032s	ERROR		UNKNOWN	0.383s
CSLL-NT	Non-terminating CSLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.003s	0.001s	0.011s	ERROR		UNKNOWN	0.843s
ADVANCED STRUCTURES										
DLL-NEXT	Forward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.034s	0.518s	0.036s	$\mathcal{O}(n)$	5.954s	UNKNOWN	0.657s
DLL-PREV	Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.031s	0.181s	0.044s	$\mathcal{O}(n)$	6.459s	UNKNOWN	0.712s
DLL-NT	Non-terminating DLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.011s	0.004s	0.024s	ERROR		UNKNOWN	0.684s
DLL-INT	Forward DLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.044s	0.654s	0.044s	$\mathcal{O}(n)$	5.723s	UNKNOWN	0.946s
DLL-PAR	Parallel Forward and Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.058s	0.510s	0.069s	ERROR		UNKNOWN	0.668s
BUTTERFLY	Terminating Butterfly Loop	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.005s	0.054s	0.024s	$\mathcal{O}(n)$	7.389s	$\mathcal{O}(n)$	0.883s
BUTTERFLY-INT	Terminating Butterfly Loop with int combination	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.026s	0.198s	0.059s	$\mathcal{O}(n)^*$	3.513s	UNKNOWN	0.899s
BUTTERFLY-NT	Non-terminating Butterfly Loop	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.005s	0.090s	0.015s	$\mathcal{O}(n)^*$	7.768s	UNKNOWN	1.701s
BST-DOUBLE	Leftmost BST Traversal with nested Rightmost	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	25.147s	12.523s	0.203s	$\mathcal{O}(n^2)^{**}$	14.547s	UNKNOWN	3.004s
BST-LEFT	Leftmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.947s	7.321s	0.171s	$\mathcal{O}(n)^{**}$	13.335s	UNKNOWN	2.476s
BST-RIGHT	Rightmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.895s	5.779s	0.168s	$\mathcal{O}(n)^{**}$	13.007s	UNKNOWN	2.457s
BST-LR	Random BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.331s	7.010s	0.188s	$\mathcal{O}(n)^{**}$	14.488s	UNKNOWN	2.619s
2-LVL SL-L1	2-lvl Skip-list Traversal via lvl1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.309s	0.837s	0.036s	ERROR		UNKNOWN	1.449s
2-LVL SL-L2	2-lvl Skip-list Traversal via lvl2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.096s	0.526s	0.042s	ERROR		UNKNOWN	1.442s
ADVANCED ALGORITHMS										
FUNCQUEUE	Queue implemented by two SLLs	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.046s	0.519s	0.136s	$\mathcal{O}(n)$	8.222s	UNKNOWN	4.808s
PARTITIONS	SLL Partitioning (from Sec 2)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.094s	0.729s	0.059s	$\mathcal{O}(n^2)$	8.526s	$\mathcal{O}(n^2)$	7.047s
INSERTSORT	Insert Sort on SLL	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.041s	0.288s	0.051s	$\mathcal{O}(n^2)$	6.453s	$\mathcal{O}(n^2)$	0.904s
MERGEINNER	Showcase example of Atkey [4]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.589s	14.080s	1.502s	$\mathcal{O}(n^2)$	57.935s	TIMEOUT(60s)	

STRUCTURES category, we infer bounds for programs on more complex structures — Binary Trees (BST), Double-Linked Lists (DLL), and even 2-level skip-lists (2-LVL SL). The last category ADVANCED ALGORITHMS includes experiments with various more advanced algorithms, including show cases taken from related work.

In benchmarks marked with (*), APROVE returned an incorrect bound in our experiments. Further, in benchmarks marked with (**), we obtained different bounds from different runs of APROVE even though it was run in exactly the same way. In both cases, we were unable to find the reason. Moreover, we remark that while the measured times show that RANGER is mostly faster, the measured times of APROVE and COSTA may be biased by using different target machines and implementations of the benchmarks (C vs Java).

The results confirm that our approach, conceived as highly parametric in the underlying shape and bounds analyses, allowed us to successfully combine an advanced shape analysis with a state-of-the-art implementation of amortized resource bounds analysis. Due to this, we were able to fully automatically derive tight complexity bounds even over data structures such as 2-level skip-lists, which are challenging even for safety analysis, and to get more precise and tight bounds for algorithms like PARTITIONS or FUNCQUEUE, which require amortized reasoning to get the precise bound. The most encouraging result is the fully automatically computed precise linear bound for the `mergeInner` method [4]. While APROVE was able to process the example, it was still not able to infer the precise interplay between the traversals of the involved SLL partitions and numeric values needed to compute the precise linear bound.

Of course, our path-based norms do have their limitations too. They are, e.g., not sufficient to verify algorithms like the Deutsch-Schorr-Waite tree traversal algorithm or tree destruction algorithms, which could probably be verified using size-based norms, based on counting all memory locations reachable from a given location. We thus see an approach combining such norms (perhaps with suitably bounded scope) with our norms as an interesting direction of future research along with a better support of norms based on data stored in dynamic data structures.

References

1. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. J. Autom. Reasoning **59**(1) (2017) 3–45
2. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Proc. of POPL'09. (2009) 239–251
3. Hofmann, M., Rodriguez, D.: Automatic type inference for amortised heap-space analysis. In: Proc. of ESOP'13. Number 7792 in LNCS, Springer (2013)
4. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science **7**(2) (2011)
5. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Automatic Inference of Resource Consumption Bounds. In: Proc. of LPAR-18. Volume 7180 of LNCS., Springer
6. Frohn, F., Giesl, J.: Complexity analysis for java with aprove. In: Proc of IFM'17. (2017) 85–101
7. Holik, L., Hruska, M., Lengal, O., Rogalewicz, A., Simacek, J., Vojnar, T.: Forester: Shape analysis using tree automata (competition contribution). In: Proc. of TACAS'15. Volume 9035 of LNCS., Springer (2015)
8. Sinn, M., Zuleger, F.: Loopus - a tool for computing loop bounds for c programs. In: Proc. of WING@ETAPS/IJCAR. (2010)
9. Ströder, T., Aschermann, C., Frohn, F., Hensel, J., Giesl, J.: Aprove: Termination and memory safety of c programs (competition contribution). In: Proc. of TACAS'15. LNCS, Springer
10. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists are Counter Automata. Formal Methods in System Design **38**("2) (2011)
11. Distefano, D., Berdine, J., Cook, B., O'Hearn, P.: Automatic Termination Proofs for Programs with Shape-shifting Heaps. In: Proc. of CAV'06. Volume 4144 of LNCS., Springer
12. Lahiri, S., Qadeer, S.: Verifying Properties of Well-Founded Linked Lists. In: Proc. of POPL'06, ACM Press (2006)
13. Yavuz-Kahveci, T., Bultan, T.: Automated verification of concurrent linked lists with counters. In: Proc. of SAS'02. Volume 2477 of LNCS., Springer 69–84
14. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. In: Proc. of POPL'07, ACM
15. Rugina, R.: Shape analysis quantitative shape analysis. In: Proc. of SAS'04. Volume 3148 of LNCS., Springer (2004)
16. Habermehl, P., Iosif, R., Rogalewicz, A., Vojnar, T.: Proving Termination of Tree Manipulating Programs. In: Proc. of ATVA'07. Volume 4762 of LNCS., Springer
17. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: Proc. of POPL'10
18. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Thor: A tool for reasoning about shape and arithmetic. In: Proc. of CAV'08. Volume 5123 of LNCS., Springer
19. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. (2014) 745–761
20. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract Regular Model Checking. In: Proc. of CAV'04. Volume 3114 of LNCS., Springer (2004)
21. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Proc. of TACAS'13, Springer
22. Holik, L., Hruska, M., Lengal, O., Rogalewicz, A., Vojnar, T.: Counterexample Validation and Interpolation-Based Refinement for Forest Automata. In: Proc. of VMCAI'17, Springer
23. Holik, L., Simacek, J., Rogalewicz, A., Vojnar, T.: Fully Automated Shape Analysis Based on Forest Automata. In: Proc. of CAV'13. Volume 8044 of LNCS., Springer (2013)
24. Bjørner, N.: The Z3 Theorem Prover URL: `https://github.com/Z3Prover/z3/`.

Invariant Generation for Multi-Path Loops with Polynomial Assignments

Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács*

Technische Universität Wien
Institut für Informationssysteme 184
Favoritenstraße 9-11
Vienna A-1040, Austria
ahumenbe@forsyte.at
maximilian@mjaroschek.com
lkovacs@forsyte.at

Abstract. Program analysis requires the generation of program properties expressing conditions to hold at intermediate program locations. When it comes to programs with loops, these properties are typically expressed as loop invariants. In this paper we study a class of multi-path program loops with numeric variables, in particular nested loops with conditionals, where assignments to program variables are polynomial expressions over program variables. We call this class of loops *extended P-solvable* and introduce an algorithm for generating all polynomial invariants of such loops. By an iterative procedure employing Gröbner basis computation, our approach computes the polynomial ideal of the polynomial invariants of each program path and combines these ideals sequentially until a fixed point is reached. This fixed point represents the polynomial ideal of all polynomial invariants of the given extended P-solvable loop. We prove termination of our method and show that the maximal number of iterations for reaching the fixed point depends linearly on the number of program variables and the number of inner loops. In particular, for a loop with m program variables and r conditional branches we prove an upper bound of $m \cdot r$ iterations. We implemented our approach in the Aligator software package. Furthermore, we evaluated it on 18 programs with polynomial arithmetic and compared it to existing methods in invariant generation. The results show the efficiency of our approach.

1 Introduction

Reasoning about programs with loops requires loop invariants expressing properties that hold before and after every loop iteration. The difficulty of generating such properties automatically comes from the use of non-linear arithmetic,

* All authors are supported by the ERC Starting Grant 2014 SYMCAR 639270. Furthermore, we acknowledge funding from the Wallenberg Academy Fellowship 2014 TheProSE, the Swedish VR grant GenPro D0497701, and the Austrian FWF research project RiSE S11409-N23. We also acknowledge support from the FWF project W1255-N23.

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 226–246, 2018.
https://doi.org/10.1007/978-3-319-73721-8_11

unbounded data structures, complex control flow, just to name few of the reasons. In this paper we focus on multi-path loops with numeric variables and polynomial arithmetic and introduce an automated approach inferring *all* loop invariants as polynomial equalities among program variables. For doing so, we identify a class of multi-path loops with nested conditionals, where assignments to program variables are polynomial expressions over program variables. Based on our previous work [4], we call this class of loops *extended P-solvable*. Compared to [4] where only single-path programs with polynomial arithmetic were treated, in this paper we generalize the notion of extended P-solvable loops to multi-path loops; single-path loops being thus a special case of our method.

For the class of extended P-solvable loops, we introduce an automated approach computing all polynomial invariants. Our work exploits the results of [17,9] showing that the set of polynomial invariants forms a polynomial ideal, called the polynomial invariant ideal. Hence, the task of generating all polynomial invariants reduces to the problem of generating a basis of the polynomial invariant ideal. Following this observation, given an extended P-solvable loop with nested conditionals, we proceed as follows: we (i) turn the multi-path loop into a sequence of single-path loops, (ii) generate the polynomial invariant ideal of each single-path loop and (iii) combine these ideals iteratively until the polynomial invariant ideal of the multi-path loop is derived.

A crucial property of extended P-solvable loops is that the single-path loops corresponding to one path of the multi-path loop are also extended P-solvable. For generating the polynomial invariant ideal of extended P-solvable single-path loops, we model loops by a system of algebraic recurrences, compute the closed forms of these recurrences by symbolic computation as described in [4] and compute the Gröbner basis of the polynomial invariant ideal from the system of closed forms. When combining the polynomial invariant ideals of each extended P-solvable single-path loop, we prove that the "composition" maintains the properties of extended P-solvable loops. Further, by exploiting the algebraic structures of the polynomial invariant ideals of extended P-solvable loops, we prove that the process of iteratively combining the polynomial invariant ideals of each extended P-solvable single-path loop is finite. That is, a fixed point is reached in a finite number of steps. We prove that this fixed point is the polynomial invariant ideal of the extended P-solvable loop with nested conditionals. We also show that reaching the fixed point depends linearly on the number of program variables and the number of inner loops. In particular, for a loop with m program variables and r inner loops (paths) we prove an upper bound of $m \cdot r$ iterations. The termination proof of our method implies the completeness of our approach: for an extended P-solvable loop with nested conditionals, our method computes all its polynomial invariants. This result generalizes and corrects the result of [10] on programs for more restricted arithmetic than extended P-solvable loops. Our class of programs extends the programming model of [10] with richer arithmetic and our invariant generation procedure also applies to [10]. As such, our proof of termination also yields a termination proof for [10].

We implemented our approach in the open source Mathematica package Aligator and evaluated our method on 18 challenging examples. When compared to state-of-the-art tools in invariant generation, Aligator performed much better in 14 examples out of 18.

The paper is organized as follows: We start by giving the necessary details about our programming model in Section 2.1 and provide background about polynomial rings and ideals in Section 2.2. In Section 3.1 we recall the notion of extended P-solvable loops from [4]. The lemmas and propositions of Section 3.2 will then help us to prove termination of our invariant generation procedure in Section 3.3. Finally, Section 4 describes our implementation in Aligator, together with an experimental evaluation of our approach.

Related Work. Generation of non-linear loop invariants has been addressed in previous research. We discuss here some of the most related works that we are aware of.

The methods of [11,18] compute polynomial equality invariants by fixing an a priori bound on the degree of the polynomials. Using this bound, a template invariant of fixed degree is constructed. Properties of polynomial invariants, e.g. inductiveness, are used to generate constraints over the unknown coefficients of the template coefficients and these constraints are then solved in linear or polynomial algebra. An a priori fixed polynomial degree is also used in [16,2]. Unlike these approaches, in our work we do not fix the degree of polynomial invariants but generate all polynomial invariants (and not just invariants up to a fixed degree). Our restrictions come in the programming model, namely treating only loops with nested conditionals and polynomial arithmetic. For such programs, our approach is complete.

Another line of research uses abstract interpretation in conjunction with recurrence solving and/or polynomial algebra. The work of [17] generates all polynomial invariants of so-called *simple loops* with nested conditionals. The approach combines abstract interpretation with polynomial ideal theory. Our model of extended P-solvable loops is much more general than simple loops, for example we allow multiplication with the loop counter and treat algebraic, and not only rational, numbers in closed form solutions. Abstract interpretation is also used in [3,12,7] to infer non-linear invariants. The programming model of these works handle loops whose assignments induce linear recurrences with constant coefficients. Extended P-solvable loops can however yield more complex recurrence equations. In particular, when comparing our work to [7], we note that the recurrence equations of program variables in [7] correspond to a subclass of linear recurrences with constant coefficients: namely, recurrences whose closed form representations do not include non-rational algebraic numbers. Our work treats the entire class of linear recurrences with constant coefficients and even handles programs whose arithmetic operations induce a class of linear recurrences with polynomial coefficients in the loop counter. While the non-linear arithmetic of our work is more general than the one in [7], we note that the programming model of [7] can handle programs that are more complex than the

ones treated in our work, in particular due to the presence of nested loops and function/procedure calls. Further, the invariant generation approach of [7] is property-guided: invariants are generated in order to prove the safety assertion of the program. Contrarily to this, we generate all invariants of the program and not only the ones implying the safety assertion.

Solving recurrences and computing polynomial invariant ideals from a system of closed form solution is also described in [9]. Our work builds upon the results of [9] but generalizes [9] to extended P-solvable loops. Moreover, we also prove that our invariant generation procedure terminates. Our termination result generalizes [10] by handling programs with more complex polynomial arithmetic. Furthermore, instead of computing the invariant ideals of all permutations of a given set of inner loops and extending this set until a polynomial ideal as a fixed point is reached, we generate the polynomial invariant ideal of just one permutation iteratively until we reach the fixed point. As a result we have to perform less Gröbner basis computations in the process of invariant generation.

A data-driven approach to invariant generation is given in [20], where concrete program executions are used to generate invariant candidates. Machine learning is then used to infer polynomial invariants from the candidate ones. In our work we do not use invariant candidates. While the program flow in our programming model is more restricted then [20], to the best of our knowledge, none of the above cited methods can fully handle the polynomial arithmetic of extended P-solvable loops.

2 Preliminaries

2.1 Programming Model and Invariants

Let $\mathbb{K}$ be a computable field of characteristic zero. This means that addition and multiplication can be carried out algorithmically, that there exists an algorithm to test if an element in $\mathbb{K}$ is zero, and that the field of rational numbers $\mathbb{Q}$ is a subfield of $\mathbb{K}$. For variables $x_1, \ldots, x_n$, the ring of multivariate polynomials over $\mathbb{K}$ is denoted by $\mathbb{K}[x_1, \ldots, x_n]$, or, if the number of variables is clear from (or irrelevant in) the context, by $\mathbb{K}[\boldsymbol{x}]$. Correspondingly, $\mathbb{K}(x_1, \ldots, x_m)$ or $\mathbb{K}(\boldsymbol{x})$ denotes the field of rational functions over $\mathbb{K}$ in $x_1, \ldots, x_m$. If every polynomial in $\mathbb{K}[x]$ with a degree ≥ 1 has at least one root in $\mathbb{K}$, then $\mathbb{K}$ is called algebraically closed. An example for such a field is $\overline{\mathbb{Q}}$, the field of algebraic numbers. In contrast, the field of complex numbers $\mathbb{C}$ is algebraically closed, but not computable, and $\mathbb{Q}$ is computable, but not algebraically closed. We suppose that $\mathbb{K}$ is always algebraically closed. This is not necessary for our theory, as we only need the existence of roots for certain polynomials, which is achieved by choosing $\mathbb{K}$ to be an appropriate algebraic extension field of $\mathbb{Q}$. It does, however, greatly simplify the statement of our results.

In our framework, we consider a program B to be a loop of the form

while ... **do**
  B' (1)
end while

where B' is a program block that is either the empty block ϵ, an assignment $v_i = f(v_1, \dots, v_m)$ for a rational function $f \in \mathbb{K}(x_1, \dots, x_m)$ and program variables $v_1, \dots, v_m$, or has one of the composite forms

sequential	inner loop	conditional

$B_1; B_2$	**while** ... **do**
	B_1
	end while
		if ... **then**
		B_1 **else** B_2
		end if

for some program blocks B_1 and B_2 and the usual semantics. We omit conditions for the loop and if statements, as the problem of computing all polynomial invariants is undecidable when taking affine equality tests into account [11]. Consequently, we regard loops as non-deterministic programs in which each block of consecutive assignments can be executed arbitrarily often. More precisely, grouping consecutive assignments into blocks $B_1, \dots, B_r$, any execution path of B can be written in the form

$$B_1^{n_1}; B_2^{n_2}; \dots; B_r^{n_r}; B_1^{n_{r+1}}; B_2^{n_{r+2}}; \dots$$

for a sequence $(n_i)_{i \in \mathbb{N}}$ of non-negative integers with finitely many non-zero elements. To that effect, we interpret any given program (1) as the set of its execution paths, written as

$$B = (B_1^*; B_2^*; \dots; B_r^*)^*.$$

We adapt the well-established Hoare triple notation

$$\{P\}B\{Q\}, \tag{2}$$

for program specifications, where P and Q are logical formulas, called the pre- and postcondition respectively, and B is a program. In this paper we focus on partial correctness of programs, that is a Hoare triple (2) is correct if every terminating computation of B which starts in a state satisfying P terminates in a state that satisfies Q.

In this paper we are concerned with computing polynomial invariants for a considerable subset of loops of the form (1). These invariants are algebraic dependencies among the loop variables that hold after any number of loop iterations.

Definition 1. *A polynomial $p \in \mathbb{K}[x_1, \dots, x_m]$ is a polynomial loop invariant for a loop $B = B_1^*; \dots; B_r^*$ in the program variables $v_1, \dots, v_m$ with initial values $v_1(0), \dots, v_m(0)$, if for every sequence $(n_i)_{i \in \mathbb{N}}$ of non-negative integers with finitely many non-zero elements, the Hoare triple*

$$\{p(v_1, \dots, v_m) = 0 \wedge \bigwedge_{i=0}^{m} v_i = v_i(0)\}$$

$$B_1^{n_1}; B_2^{n_2} \dots, B_r^{n_r}; B_1^{n_{r+1}}; \dots$$

$$\{p(v_1, \dots, v_m) = 0\}$$

is correct.

2.2 Polynomial Rings and Ideals

Polynomial invariants are algebraic dependencies among the values of the variables at each loop iteration. Obviously, non-trivial dependencies do not always exist.

Definition 2. *Let* $\mathbb{L} / \mathbb{K}$ *be a field extension. Then* $a_1, \ldots, a_n \in \mathbb{L}$ *are* algebraically dependent *over* $\mathbb{K}$ *if there exists a* $p \in \mathbb{K}[x_1, \ldots, x_n] \setminus \{0\}$ *such that* $p(a_1, \ldots, a_n) = 0$. *Otherwise they are called* algebraically independent.

In [8,17], it is observed that the set of all polynomial loop invariants for a given loop forms an ideal. It is this fact that facilitates all of our subsequent reasoning.

Definition 3. *A subset* $\mathcal{I}$ *of a commutative ring* R *is called an* ideal, *written* $\mathcal{I} \lhd R$, *if it satisfies the following three properties:*

1. $0 \in \mathcal{I}$.
2. *For all* $a, b \in \mathcal{I}$: $a + b \in \mathcal{I}$.
3. *For all* $a \in \mathcal{I}$ *and* $b \in R$: $a \cdot b \in \mathcal{I}$.

Definition 4. *Let* $\mathcal{I} \lhd R$. *Then* $\mathcal{I}$ *is called*

- proper *if it is not equal to* R,
- prime *if* $a \cdot b \in \mathcal{I}$ *implies* $a \in \mathcal{I}$ *or* $b \in \mathcal{I}$, *and*
- radical *if* $a^n \in \mathcal{I}$ *implies* $a \in \mathcal{I}$.

The height $\text{hg}(\mathcal{I}) \in \mathbb{N}$ *of a prime ideal* $\mathcal{I}$ *is equal to* n *if* n *is the maximal length of all possible chains of prime ideals* $\mathcal{I}_0 \subset \mathcal{I}_2 \subset \cdots \subset \mathcal{I}_n = \mathcal{I}$.

Example 5. The set of even integers $2\mathbb{Z}$ is an ideal of $\mathbb{Z}$. In general $n\mathbb{Z}$ for a fixed integer n is an ideal of $\mathbb{Z}$. It is prime if and only if n is a prime number.

Polynomial ideals can informally be interpreted as the set of all consequences when it is known that certain polynomial equations hold. In fact, if we have given a set P of polynomials of which we know that they serve as algebraic dependencies among the variables of a given loop, the ideal generated by P then contains all the polynomials that consequently have to be polynomial invariants as well.

Definition 6. *A subset* $B \subseteq \mathcal{I}$ *of an ideal* $\mathcal{I} \lhd R$ *is called a* basis *for* $\mathcal{I}$ *if*

$$\mathcal{I} = \langle B \rangle := \{a_0 b_0 + \cdots + a_m b_m \mid m \in \mathbb{N}, a_0, \ldots, a_m \in R, b_0 \ldots, b_m \in B\}.$$

We say that B generates $\mathcal{I}$.

A basis for a given ideal in a ring does not necessarily have to be finite. However, a key result in commutative algebra makes sure that in our setting we only have to consider finitely generated ideals.

Theorem 7 (Hilbert's Basis Theorem – Special case). *Every ideal in* $\mathbb{K}[\boldsymbol{x}]$ *has a finite basis.*

Subsequently, whenever we say we are given an ideal $\mathcal{I}$, we mean that we have given a finite basis of $\mathcal{I}$.

There is usually more than one basis for a given ideal and some are more useful for certain purposes than others. In his seminal PhD thesis [1], Buchberger introduced the notion of Gröbner bases for polynomial ideals and an algorithm to compute them. While, for reasons of brevity, we will not formally define these bases, it is important to note that with their help, central questions concerning polynomial ideals can be answered algorithmically.

Theorem 8. *Let* $p \in \mathbb{K}[x_1, \dots, x_n]$ *and* $\mathcal{I}, \mathcal{J} \lhd \mathbb{K}[x_1, \dots, x_n]$. *There exist algorithms to decide the following problems.*

1. *Decide if* p *is an element of* $\mathcal{I}$.
2. *Compute a basis of* $\mathcal{I} + \mathcal{J}$.
3. *Compute a basis of* $\mathcal{I} \cap \mathcal{J}$.
4. *For* $\{\tilde{x}_1, \dots \tilde{x}_m\} \subseteq \{x_1, \dots, x_n\}$, *compute a basis of* $\mathcal{I} \cap \mathbb{K}[\tilde{x}_1, \dots, \tilde{x}_m]$.
5. *Let* $q \in \mathbb{K}[\boldsymbol{x}]$. *Compute a basis for*
$$\mathcal{I} : \langle q \rangle^\infty := \{q \in \mathbb{K}[\boldsymbol{x}] \mid \exists n \in \mathbb{N} : q^n p \in \mathcal{I}\}.$$
The ideal $\mathcal{I} : \langle q \rangle^\infty$ *is called the* saturation *of* $\mathcal{I}$ *with respect to* q.

We will use Gröbner bases to compute the ideal of all algebraic relations among given rational functions. For this, we use the polynomials $q_i y_i - p_i$ to model the equations $y_i = q_i/p_i$ by multiplying the equation with the denominator. In order to model the fact that the denominator is not identically zero, and therefore allowing us to divide by it again, we use the saturation with respect to the least common multiple of all denominators. To see why this is necessary, consider $y_1 = y_2 = \frac{x_1}{x_2}$. An algebraic relation among y_1 and y_2 is $y_1 - y_2$, but with the polynomials $x_2 y_1 - x_1$ and $x_2 y_2 - x_1$, we only can derive $x_2(y_1 - y_2)$. We have to divide by x_2.

Theorem 9. *Let* $r_1, \dots, r_m \in \mathbb{K}(\boldsymbol{x})$ *and let the numerator of* r_i *be given by* $p_i \in \mathbb{K}[\boldsymbol{x}]$ *and the denominator by* $q_i \in \mathbb{K}[\boldsymbol{x}]$. *The ideal of all polynomials* p *in* $\mathbb{K}[\boldsymbol{y}]$ *with* $p(r_1, \dots, r_m) = 0$ *is given by*
$$\left(\sum_{i=1}^{m} \langle q_i y_i - p_i \rangle\right) : \langle \mathrm{lcm}(q_1, \dots, q_m) \rangle^\infty \cap \mathbb{K}[\boldsymbol{y}],$$
where $\mathrm{lcm}(\dots)$ *denotes the least common multiple.*

Proof. Write $d := \mathrm{lcm}(q_1, \dots, q_m)$. The theorem can be easily verified from the fact that, for any given p with $p(r_1, \dots, r_m) = 0$, there exists a $k \in \mathbb{N}$ such that $d^k p(r_1, \dots, r_m) = 0$ is an algebraic relation for $p_1, \dots, p_m$ (by clearing denominators in the equation $p(r_1, \dots, r_m) = 0$). $\square$

A polynomial ideal $\mathcal{I} \lhd \mathbb{K}[\boldsymbol{x}]$ gives rise to a set of points in $\mathbb{K}^n$ for which all polynomials in $\mathcal{I}$ vanish simultaneously. This set is called a *variety*.

Definition 10. *Let $\mathcal{I} \lhd \mathbb{K}[x_1, \ldots, x_n]$ be an ideal. The set*

$$V(\mathcal{I}) = \{(a_1, \ldots, a_n) \in \mathbb{K}^n \mid p(a_1, \ldots, a_n) = 0 \text{ for all } p \in \mathcal{I}\},$$

is the variety *defined by $\mathcal{I}$.*

Varieties are one of the central objects of study in algebraic geometry. Certain geometric shapes like points, lines, circles or balls can be described by prime ideals and come with an intuitive notion of a dimension, e.g. points have dimension zero, lines and circles have dimension one and balls have dimension two. The notion of the Krull dimension of a ring formalizes this intuition when being applied to the quotient ring $\mathbb{K}[\boldsymbol{x}]/\mathcal{I}$. In this paper, we will use the Krull dimension to provide an upper bound for the number of necessary iterations of our algorithm.

Definition 11. *The* Krull dimension *of a commutative ring R is the supremum of the lengths of all chains $\mathcal{I}_0 \subset \mathcal{I}_1 \subset \ldots$ of prime ideals.*

Theorem 12. *The Krull dimension of $\mathbb{K}[x_1, \ldots, x_n]$ is equal to n.*

3 Extended P-Solvable Loops

In [4] the class of *P-solvable* loops [9] was extended to so-called *extended P-solvable* loops. So far, this class captures loops with assignments only, i.e. loops without any nesting of conditionals and loops. In Section 3.3 we close this gap by introducing a new approach for computing invariants of multi-path loops which generalizes the algorithm proposed in [10]. Before dealing with multi-path loops, we recall the notion of extended P-solvable loops in Section 3.1 and showcase the invariant ideal computation.

3.1 Loops with assignments only

In this section, we restrain ourselves to loops whose bodies are comprised of rational function assignments only. This means that we restrict the valid composite forms in a program of the form (1) to sequential compositions and, for the moment, exclude inner loops and conditional branches. We therefore consider a loop $L = B_1^*$ where B_1 is a single block containing only variable assignments.

Each variable v_i in a given loop of the form (1) gives rise to a sequence $(v_i(n))_{n\in\mathbb{N}}$, where n is the number of loop iterations. The class of eligible loops is then defined based on the form of these sequences. Let $r(x)^{\underline{n}}$ denote the *falling factorial* defined as $\prod_{i=0}^{n-1} r(x-i)$ for any $r \in \mathbb{K}(x)$ and $n \in \mathbb{N}$.

Definition 13. *A loop with assignments only is called* extended P-solvable *if each of its recursively changed variables determines a sequence of the form*

$$v_i(n) = \sum_{j\in\mathbb{Z}^\ell} p_{i,j}(n, \theta_1^n, \ldots, \theta_k^n)((n+\zeta_1)^{\underline{n}})^{j_1} \cdots ((n+\zeta_\ell)^{\underline{n}})^{j_\ell} \tag{3}$$

where $k, \ell \in \mathbb{N}$, the $p_{i,j}$ are polynomials in $\mathbb{K}(x)[y_1, \dots, y_k]$, not identically zero for finitely many $j \in \mathbb{Z}^\ell$, the θ_i are elements of $\mathbb{K}$ and the ζ_i are elements of $\mathbb{K} \setminus \mathbb{Z}^-$ with $\theta_i \neq \theta_j$ and $\zeta_i - \zeta_j \notin \mathbb{Z}$ for $i \neq j$.

Definition 13 extends the class of P-solvable loops in the sense that each sequence induced by an extended P-solvable loop is the sum of a finitely many hypergeometric sequences. This comprises C-finite sequences as well as hypergeometric sequences and sums and Hadamard products of C-finite and hypergeometric sequences. In contrast, P-solvable loops induce C-finite sequences only. For details on C-finite and hypergeometric sequences we refer to [5].

Every sequence of the form (3) can be written as

$$v_j^{(1)} = r_j(\boldsymbol{v}^{(0)}, \boldsymbol{\theta}, (n + \boldsymbol{\zeta})^{\underline{n}}, n)$$

where $r_j = p_i/q_i$ is a rational function, and $v^{(0)}$ and $v^{(1)}$ denote the values of v before and after the execution of the loop. Let $I(\boldsymbol{\theta}, \boldsymbol{\zeta}) \lhd \mathbb{K}[y_0, \dots, y_{k+\ell}]$ be the ideal of all algebraic dependencies in the variables $y_0, \dots, y_{k+\ell}$ between the sequence $(n)_{n\in\mathbb{N}}$, the exponential sequences $\theta_1^n, \dots, \theta_k^n$ and the sequences $(n+\zeta_1)^{\underline{n}}, \dots, (n+\zeta_\ell)^{\underline{n}}$. Note that it was shown in [4] that this ideal is the same as the extension of the ideal $I(\boldsymbol{\theta}) \lhd \mathbb{K}[y_0, \dots, y_k]$ of all algebraic dependencies between the θ^n in $\mathbb{K}[y_0, \dots, y_k]$ to $\mathbb{K}[y_0, \dots, y_{k+\ell}]$, as the factorial sequences $(n+\zeta_i)^{\underline{n}}$ are algebraically independent from the exponential sequences θ_i^n. Now the following proposition states how the invariant ideal of an extended P-solvable loop can be computed.

Proposition 14 ([4]). *For an extended P-solvable loop with program variables $v_1, \dots, v_m$ the invariant ideal is given by*

$$\left(\left(\sum_{j=1}^{m} \langle q_j(\boldsymbol{v}^{(0)}, \boldsymbol{y}) v_j^{(1)} - p_j(\boldsymbol{v}^{(0)}, \boldsymbol{y})\rangle\right) : \langle \mathrm{lcm}(q_1, \dots, q_m)\rangle^\infty + I(\boldsymbol{\theta}, \boldsymbol{\zeta})\right) \cap \mathbb{K}[\boldsymbol{v}^{(1)}, \boldsymbol{v}^{(0)}].$$

Example 15. Consider the following loop with relevant program variables a, b and c.

while true **do**
$a := 2 \cdot (n+1)(n+\frac{3}{2}) \cdot a$
$b := 4 \cdot (n+1) \cdot b$
$c := \frac{1}{2} \cdot (n+\frac{3}{2}) \cdot c$
$n := n+1$
end while

The extracted recurrence relations admit the following system of closed form solutions:

$$a_n = 2^n \cdot a_0 \cdot (n)^{\underline{n}} \cdot (n + \frac{1}{2})^{\underline{n}},$$
$$b_n = 4^n \cdot b_0 \cdot (n)^{\underline{n}},$$
$$c_n = 2^{-n} \cdot c_0 \cdot (n + \frac{1}{2})^{\underline{n}}.$$

Since every closed form solution is of the form (3) we have an extended P-solvable loop, and we can apply Proposition 14 to compute the invariant ideal:

$$(\mathcal{I} + I(\boldsymbol{\theta}, \boldsymbol{\zeta})) \cap \mathbb{K}[a^{(1)}, b^{(1)}, c^{(1)}, a^{(0)}, b^{(0)}, c^{(0)}] = \langle b^{(1)} \cdot c^{(1)} \cdot a^{(0)} - a^{(1)} \cdot b^{(0)} \cdot c^{(0)} \rangle,$$

where

$$\begin{aligned} \mathcal{I} &= \langle a^{(1)} - y_1 \cdot a^{(0)} \cdot z_1 z_2, b^{(1)} - y_2 \cdot b^{(0)} \cdot z_1, c^{(1)} - y_3 \cdot c^{(0)} \cdot z_2 \rangle, \\ I(\boldsymbol{\theta}, \boldsymbol{\zeta}) &= \langle y_1^2 - y_2, y_1 y_3 - 1, y_2 y_3 - y_1 \rangle. \end{aligned}$$

The ideal $I(\boldsymbol{\theta}, \boldsymbol{\zeta})$ in variables y_1, y_2, y_3 is the set of all algebraic dependencies among $2^n, 4^n$ and 2^{-n}, and $\mathcal{I}$ is generated by the closed form solutions where exponential and factorial sequences are replaced by variables y_1, y_2, y_3 and z_1, z_2.

3.2 Algebraic Dependencies of Composed Rational Functions with Side Conditions

In this section we give the prerequisites for proving termination of the invariant generation method for multi-path loops (Section 3.3). The results of this section will allow us to proof termination by applying Theorem 12.

Let $\boldsymbol{v}^{(i)} = v_1^{(i)}, \ldots, v_m^{(i)}$ and $\boldsymbol{y}^{(i)} = y_1^{(i)}, \ldots, y_\ell^{(i)}$ for $i \in \mathbb{N}$. We model the situation in which the value of the jth loop variable after the execution of the ith block in (1) is given by a rational function in the $\boldsymbol{y}^{(i)}$ (which, for us, will be the exponential and factorial sequences as well as the loop counter) and the 'old' variable values $\boldsymbol{v}^{(i-1)}$ and is assigned to $v_j^{(i)}$. Set $\mathcal{I}_0 = \sum_{j=1}^m \langle v_j^{(1)} - v_j^{(0)} \rangle$ and let $I_i \lhd \mathbb{K}[\boldsymbol{y}^{(i)}]$ for $i \in \mathbb{N}^*$. Furthermore, let $q_j^{(i)}, p_j^{(i)} \in \mathbb{K}[\boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)}]$ such that for fixed i there exists a $\boldsymbol{y} \in V(I_i)$ with $p_j^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}) / q_j^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}) = \boldsymbol{v}_j^{(i)}$ for all j and with $d_i := \operatorname{lcm}(q_1^{(i)}, \ldots, q_m^{(i)})$ we have $d_i \notin I_i$ and $d_i(\boldsymbol{v_i}, \boldsymbol{y}) = 1$. Set

$$J_i = \sum_{j=1}^m \langle q_j^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)}) v_j^{(i+1)} - p_j^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)}) \rangle.$$

Remark 16. The requirement for the existence of a point $\boldsymbol{y}$ in $V(I_i)$ such that $p_j^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}) / q_j^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}) = \boldsymbol{v}_j^{(i)}$ for all j and $d_i(\boldsymbol{v_i}, \boldsymbol{y}) = 1$ is always fulfilled in our context, as it is a formalization of the fact that the execution of a loop L^* also allows that it is executed zero times, meaning the values of the program variables do not change.

In order to develop some intuition about the following, consider a list of consecutive loops $L_1; L_2; L_3; \ldots$ where each of them is extended P-solvable. Intuitively, the ideals I_i then correspond to the ideal of algebraic dependencies among the exponential and factorial sequences occurring in L_i, whereas J_i stands for the ideal generated by the closed form solutions of L_i. Moreover, the variables $v_j^{(i+1)}$ correspond to the values of the loop variables after the execution of the loop L_i. The following iterative computation then allows us to generate the invariant ideal for $L_1; L_2; L_3; \ldots$

$$\mathcal{I}_i := ((J_i + \mathcal{I}_{i-1} + I_i) : \langle d_i \rangle^\infty) \cap \mathbb{K}[\boldsymbol{v}^{(i+1)}, \boldsymbol{v}^{(0)}]$$

Now the remaining part of this section is devoted to proving properties of the ideals $\mathcal{I}_i$ which will help us to show that there exists an index k such that $\mathcal{I}_k = \mathcal{I}_{k'}$ for all $k' > k$ for a list of consecutive loops $L_1; \dots; L_r; L_1; \dots; L_r; \dots$ with $r \in \mathbb{N}$.

First note that the ideal $\mathcal{I}_i$ can be rewritten as

$$\begin{aligned} \mathcal{I}_i = \{p \in \mathbb{K}[\boldsymbol{v}^{(i+1)}, \boldsymbol{v}^{(0)}] \mid \exists q \in \mathcal{I}_{i-1}, k \in \mathbb{N} : \\ q \equiv d_i^k p(r_1^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)}), \dots, r_m^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}^{(i)}), \boldsymbol{v}^{(0)}) \pmod{I_i}\}. \end{aligned} \tag{4}$$

If I_i is radical, an equation **mod** I_i is, informally speaking, the same as substituting $\boldsymbol{y}$ with values from $V(I_i)$, so (4) translates to

$$\begin{aligned} \mathcal{I}_i = \{p \in \mathbb{K}[\boldsymbol{v}^{(i+1)}, \boldsymbol{v}^{(0)}] \mid \exists q \in \mathcal{I}_{i-1}, k \in \mathbb{N} : \\ \forall \boldsymbol{y} \in V(I_i) : q = d_i^k p(r_1^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}), \dots, r_m^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}), \boldsymbol{v}^{(0)})\}. \end{aligned} \tag{5}$$

We now get the following subset relation between two consecutively computed ideals $\mathcal{I}_i$.

Lemma 17. *If I_i is radical, then $\mathcal{I}_i \subseteq \mathcal{I}_{i-1}|_{\boldsymbol{v}^{(i-1)} \leftarrow \boldsymbol{v}^{(i)}}$.*

Proof. Let $p \in \mathcal{I}_i$. We have to show that there is an $r \in \mathcal{I}_{i-2}$ and a $k \in \mathbb{N}$ such that

$$r \equiv d_{i-1}^k p(r_1^{(i-1)}(\boldsymbol{v}^{(i-1)}, \boldsymbol{y}^{(i-1)}), \dots, r_m^{(i-1)}(\boldsymbol{v}^{(i-1)}, \boldsymbol{y}^{(i-1)}), \boldsymbol{v}^{(0)}) \pmod{I_{i-1}}.$$

Since I_i is radical, there is a $q \in \mathcal{I}_{i-1}$, a $z \in \mathbb{N}$, and a $\boldsymbol{y} \in V(I_i)$ with

$$q = d_i^z p(r_1^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}), \dots, r_m^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{y}), \boldsymbol{v}^{(0)}) = p(\boldsymbol{v}^{(i)}, \boldsymbol{v}^{(0)}).$$

Then, by Equation (4) for $\mathcal{I}_{i-1}$, there is an $r \in \mathcal{I}_{i-2}$ with the desired property. □

For prime ideals, we get an additional property:

Lemma 18. *If $\mathcal{I}_{i-1}$ and I_i are prime, then so is $\mathcal{I}_i$.*

Proof. Let $a \cdot b \in \mathcal{I}_i$ and denote by $a|_r$ and $b|_r$ the rational functions where each $v_j^{(i+1)}$ is substituted by $r_j^{(i)}$ in a, b respectively. Then there is a $q \in \mathcal{I}_{i-1}$ and a $k = k_1 + k_2 \in \mathbb{N}$ with $d_i^{k_1} a|_r, d_i^{k_2} b|_r \in \mathbb{K}[\boldsymbol{v}^{(i+1)}, \boldsymbol{v}^{(0)}]$

$$q \equiv d_i^k (a \cdot b)|_r \equiv d_i^{k_1} a|_r \cdot d_i^{k_2} b|_r \pmod{I_i}$$

If $d_i^k a|_r$ is zero modulo I_i, then a is an element of $\mathcal{I}_i$, as $0 \in \mathcal{I}_{i-1}$. The same argument holds for b. Suppose that $d_i^{k_1} a|_r, d_i^{k_2} b|_r \not\equiv 0 \pmod{I_i}$. Then, since I_i is prime, $\mathbb{K}[\boldsymbol{y}^{(i)}]/I_i$ is an integral domain, and so it follows that $q \not\equiv 0 \pmod{I_i}$. Now, because $\mathcal{I}_{i-1}$ is prime, it follows without loss of generality that $d_i^{k_1} a|_r \in \mathcal{I}_{i-1}$, from which we get $a \in \mathcal{I}_i$. □

We now use Lemmas 17 and 18 to give details about the minimal decomposition of $\mathcal{I}_i$.

Proposition 19. *For fixed $i_0 \in \mathbb{N}$, let all I_i, $0 \leq i \leq i_0$ be radical and let $\mathcal{I}_{i_0} = \bigcap_{k=0}^n P_k$ be the minimal decomposition of $\mathcal{I}_{i_0}$. Then*

1. *for each k there exist prime ideals $I_{k,1}, I_{k,2}, \dots$ such that P_k is equal to a $\mathcal{I}_{k,i_0}$ constructed as above with $J_1, \dots, J_{i_0}$ and $I_{k,1}, \dots, I_{k,i_0}$.*
2. *if I_{i_0+1} is radical and $\mathcal{I}_{i_0+1} = \bigcap_{j=0}^{n'} P'_j$ is the minimal decomposition of $\mathcal{I}_{i+1}$, then, for each P'_j there exists a P_k such that $P'_j \subseteq P_k|_{\boldsymbol{v}^{(i_0)} \leftarrow \boldsymbol{v}^{(i_0+1)}}$.*

Proof. We prove 1. by induction. For $i_0 = 0$, there is nothing to show. Now assume the claim holds for some $i_0 \in \mathbb{N}$ and let $I_{i_0+1} = \bigcap_{j=0}^{w} Q_j$ be the minimal decomposition of I_{i_0+1}. With this we get

$$\begin{aligned}
\mathcal{I}_{i_0+1} &= (J_{i_0+1} + \mathcal{I}_{i_0} + I_{i_0+1}) : \langle d_{i_0+1} \rangle^\infty \cap \mathbb{K}[\boldsymbol{v}^{(i_0+1)}, \boldsymbol{v}^{(0)}] \\
&= \left(\bigcap_{k=0}^{n} J_{i_0+1} + P_k + \bigcap_{j=0}^{w} Q_j \right) : \langle d_{i_0+1} \rangle^\infty \cap \mathbb{K}[\boldsymbol{v}^{(i+1)}, \boldsymbol{v}^{(0)}] \\
&= \left(\bigcap_{k=0}^{n} \bigcap_{j=0}^{w} \underbrace{(J_{i_0+1} + P_k + Q_j) : \langle d_{i_0+1} \rangle^\infty \cap \mathbb{K}[\boldsymbol{v}^{(i_0+1)}, \boldsymbol{v}^{(0)}]}_{\tilde{I}_{k,j}} \right).
\end{aligned}$$

By the induction hypothesis, each P_k admits a construction as above, and thus so does $\tilde{I}_{k,j}$. By Lemma 18, $\tilde{I}_{k,j}$ is prime. This shows 1. The second claim then follows from the fact that the prime ideals in the minimal decomposition of $\mathcal{I}_{i_0+1}$ are obtained from the P_k via J_{i_0+1} and Q_j. Since the Q_j are prime, they are also radical, and the claim follows from Lemma 17. □

3.3 Loops with conditional branches

In this section, we extend the results of Section 3.1 to loops with conditional branches. Without loss of generality, we define our algorithm for a loop of the form

while ... **do** $L_1; L_2; \dots; L_r$ **end while**

where $L_i = B_i^*$ and B_i is a block containing variable assignments only.

Let $I(\boldsymbol{\theta}_i, \boldsymbol{\zeta}_i)$ denote the ideal of all algebraic dependencies as described in Section 3.1 for a inner loop L_i. As every inner loop provides its own loop counter, we have that the exponential and factorial sequences of distinct inner loops are algebraically independent. Therefore $I(\boldsymbol{\theta}, \boldsymbol{\zeta}) := \sum_{i=0}^{r} I(\boldsymbol{\theta}_i, \boldsymbol{\zeta}_i)$ denotes the set of all algebraic dependencies between exponential and factorial sequences among the inner loops $L_1, \dots, L_r$.

Consider loop bodies $B_1, \dots, B_r$ with common loop variables $v_1, \dots, v_m$. Suppose the closed form of v_j in the ith loop body is given by a rational function in $m + k + \ell + 1$ variables:

$$v_j^{(i+1)} = r_j^{(i)}(\boldsymbol{v}^{(i)}, \boldsymbol{\theta}^n, (n + \boldsymbol{\zeta})^{\underline{n}}, n),$$

where $v_j^{(i)}$ and $v_j^{(i+1)}$ are variables for the value of v_j before and after the execution of the loop body. Then we can compute the ideal of all polynomial invariants of the non-deterministic program $(B_1^*; B_2^*; \dots; B_r^*)^*$ with Algorithm 1.

Algorithm 1 Invariant generation via fixed point computation

Input: Loop bodies $B_1, \dots, B_r$ as described.
Output: The ideal of all polynomial invariants of $(B_1^*; B_2^*; \dots; B_r^*)^*$.

1: Compute $I := I(\boldsymbol{\theta}, \boldsymbol{\zeta})$ as described above
2: $\mathcal{I}_{old} = \{0\}$, $\mathcal{I}_{new} = \sum_{j=1}^{m} \langle v_j^{(1)} - v_i^{(0)} \rangle$, $j = 0$
3: **WHILE** $\mathcal{I}_{old}|_{\boldsymbol{v}^{((j-1)\cdot r+1)} \leftarrow \boldsymbol{v}^{(j\cdot r+1)}} \neq \mathcal{I}_{new}$ **AND** $\mathcal{I}_{new} \neq \{0\}$ **DO**
4: $\quad \mathcal{I}_{old} \leftarrow \mathcal{I}_{new}$, $j \leftarrow j + 1$
5: $\quad$ **FOR** $i = 1, \dots, r$ **DO**
6: $\quad\quad \mathcal{I}_{new} \leftarrow (J_{i\cdot j} + \mathcal{I}_{old} + I) \cap \mathbb{K}[\boldsymbol{v}^{(i\cdot j+1)}, \boldsymbol{v}^{(0)}]$
7: **RETURN** $\mathcal{I}_{new}$

Lemma 20. *$I(\boldsymbol{\theta}, \boldsymbol{\zeta})$ is a radical ideal.*

Proof. The elements of $I(\boldsymbol{\theta})$ represent C-finite sequences, i.e. sequences of the form

$$f_1(n)\theta_1^n + \dots + f_k^n \theta_k^n,$$

for univariate polynomials $f_1, \dots, f_k \in \mathbb{K}[y_0]$ and pairwise distinct $\theta_1, \dots, \theta_k \in \mathbb{K}$. The claim is then proven by the fact that the Hadamard-product $a^2(n, a(0))$ of a C-finite sequence $a(n, a(0))$ with itself is zero if and only if $a(n, a(0))$ is zero, and $I(\boldsymbol{\theta}, \boldsymbol{\zeta})$ is the extension of $I(\boldsymbol{\theta})$ to $\mathbb{K}[y_0, \dots, y_{k+\ell}]$. □

Theorem 21. *Algorithm 1 is correct and terminates.*

Proof. The algorithm iteratively computes the ideals $\mathcal{I}_1, \mathcal{I}_2, \dots$ as in Section 3.2, so we will refer to I_{old} and I_{new} as $\mathcal{I}_i$ and $\mathcal{I}_{i+1}$.

Termination: $\mathcal{I}_0$ is a prime ideal of height m. Suppose after an execution of the outer loop, the condition $\mathcal{I}_i|_{\boldsymbol{v}^{(i)} \leftarrow \boldsymbol{v}^{(i+1)}} \neq \mathcal{I}_{i+1}$ holds. As $I(\boldsymbol{\theta}, \boldsymbol{\zeta})$ is radical by Lemma 20, we then get $\mathcal{I}_{i+1} \subset \mathcal{I}_i|_{\boldsymbol{v}^{(i)} \leftarrow \boldsymbol{v}^{(i+1)}}$ by Lemma 17. Thus there is a $p \in \mathbb{K}[\boldsymbol{v}^{(i+1)}, \boldsymbol{v}^{(0)}]$ with $p \in \mathcal{I}_i|_{\boldsymbol{v}^{(i)} \leftarrow \boldsymbol{v}^{(i+1)}}$ and $p \notin \mathcal{I}_{i+1}$. Then, by Proposition 19, all prime ideals P_k in the minimal decomposition of $\mathcal{I}_{i+1}$ are have to be subsets of the prime ideals in the minimal decomposition of $\mathcal{I}_i|_{\boldsymbol{v}^{(i)} \leftarrow \boldsymbol{v}^{(i+1)}}$, where at least one of the subset relations is proper. Since $p \notin \mathcal{I}_{i+1}$, the height of at least one P_k has to be reduced. The height of each prime ideal is bounded by the height of $\mathcal{I}_0$.

Correctness: Let $i \in \mathbb{N}$ be fixed and denote by $I(B; i) \lhd \mathbb{K}[\boldsymbol{v}^{(i+1)}, \boldsymbol{v}^{(0)}]$ the ideal of all polynomial invariants for the non-deterministic program

$$(B_1^*; \dots; B_r^*)^{i/r}; B_1^*; \dots; B_{i \,\mathbf{rem}\, r}^*.$$

It suffices to show that $\mathcal{I}_i$ is equal to $I(B; i)$. In fact, after i_0 iterations with $\mathcal{I}_{i_0} = \mathcal{I}_{i_0+1} = \mathcal{I}_{i_0+2} = \dots$, it follows that $\mathcal{I}_{i_0}$ is the ideal of polynomial invariants for $(B_1^*; \dots; B_r^*)^*$. Let $p \in I(B; i)$. The value of the program variable v_j in the program $B_1^*; \dots; B_{i \,\mathbf{rem}\, r}^*$ is given as the value of a composition of the closed forms of each B_k:

$$v_j = p_j^{(i)}\left(p^{(i-1)}\left(\dots (p^{(1)}(\boldsymbol{v}^{(0)}, \boldsymbol{s}_{n_1}), \dots), \boldsymbol{s}_{n_{i-1}}\right), \boldsymbol{s}_{n_i}\right),$$

with $\boldsymbol{s}_n = n, \boldsymbol{\theta}^n, (n+\boldsymbol{\zeta})^{\underline{n}}$ and $n_1, \dots, n_i \in \mathbb{N}$. The correctness then follows from the fact that that $\mathcal{I}_i$ is the ideal of all such compositions under the side condition that $(\boldsymbol{\theta}^n, (n+\boldsymbol{\zeta})^{\underline{n}}, n) \in V(I(\boldsymbol{\theta}, \boldsymbol{\zeta}))$ for any $n \in \mathbb{N}$. □

Revisiting the subset relations of the prime ideals in the minimal decomposition of $\mathcal{I}_0, \mathcal{I}_1, \dots$ gives an upper bound for the necessary number of iterations in the algorithm.

Corollary 22. *Algorithm 1 terminates after at most m iterations of the while-loop at line 3.*

Proof. Suppose the algorithm terminates after k_0 iterations of the outer loop. We look at the ideals $\mathcal{I}_{r\cdot k}$, $k \in \{0, \dots, k_0\}$. For a prime ideal P in the minimal decomposition of any $\mathcal{I}_{r\cdot(k+1)}$, there is a prime ideal Q in the minimal decomposition of $\mathcal{I}_{r\cdot k}$ such that $P \subseteq Q$. If $P = Q$, then P is a prime ideal in the minimal decomposition of each $\mathcal{I}_{r\cdot(k')}$, $k' > k$. This holds because there are only r many J_i. So if Q does not get replaced by smaller prime ideals in $\mathcal{I}_{r\cdot k+1}, \mathcal{I}_{r\cdot k+2} \dots, \mathcal{I}_{r\cdot(k+1)}$, it has to be part of the minimal decomposition for any subsequent $\mathcal{I}_i$. From this it follows that, for each k, there is a prime ideal P_k in the minimal decomposition in $\mathcal{I}_{r\cdot k}$, such that $P_0 \supset P_1 \supset \dots \supset P_{k_0}$ is a chain of proper superset relations, which then proves the claim since the height of $P_0 = \mathcal{I}_0$ is m. □

Example 23. Consider a multi-path loop L

while ... **do** $L_1; L_2$ **end while**

containing the following nested loops L_1 and L_2 and the corresponding closed form solutions:

while ... **do**		**while** ... **do**	
$a := a - b$	$a_n = a_0 - nb_0$	$b := b - a$	$b_m = b_0 - ma_0$
$p := p - q$	$p_n = p_0 - nq_0$	$q := q - p$	$q_m = q_0 - mp_0$
$r := r - s$	$r_n = r_0 - ns_0$	$s := s - r$	$s_m = s_0 - mr_0$
end while		**end while**	

For simplicity we chose inner loops without algebraic dependencies, i.e. I at line 1 will be the zero ideal and we therefore neglect it in the following computation. Moreover, we write a_i instead of $a^{(i)}$. We start with

$$\mathcal{I}_0 = \langle a_1 - a_0, b_1 - b_0, p_1 - p_0, q_1 - q_0, r_1 - r_0, s_1 - s_0 \rangle$$

followed by the first loop iteration:

$$\begin{aligned}\mathcal{I}_1 &= (J_1 + \mathcal{I}_0) \cap \mathbb{K}[a_0, b_0, p_0, q_0, r_0, s_0, a_2, b_2, p_2, q_2, r_2, s_2] \\ &= \langle b_0 - b_2, q_0 - q_2, s_0 - s_2, -p_0 s_2 + p_2 s_2 + q_2 r_0 - q_2 r_2, \\ &\qquad a_0 s_2 - a_2 s_2 - b_2 r_0 + b_2 r_2, a_0 q_2 - a_2 q_2 - b_2 p_0 + b_2 p_2 \rangle\end{aligned}$$

where

$$J_1 = \langle a_2 - a_1 + b_1 n, p_2 - p_1 + q_1 n, r_2 - r_1 + s_1 n, b_2 - b_1, q_2 - q_1, s_2 - s_1 \rangle$$

The following ideal $\mathcal{I}_2$ is then the invariant ideal for the first iteration of the outer loop L.

$$\begin{aligned}
\mathcal{I}_2 &= (J_2 + \mathcal{I}_1) \cap \mathbb{K}[a_0, b_0, p_0, q_0, r_0, s_0, a_3, b_3, p_3, q_3, r_3, s_3] \\
&= \langle -p_0 r_3 s_0 + p_3 r_3 s_3 + p_3 r_0 s_0 - p_3 r_0 s_3 - q_3 r_3^2 + q_3 r_0 r_3, \\
&\quad - p_3 s_0 + p_3 s_3 + q_0 r_3 - q_3 r_3, -p_0 s_0 + p_3 s_3 + q_0 r_0 - q_3 r_3, \\
&\quad a_3 s_0 - a_3 s_3 - b_0 r_3 + b_3 r_3, a_0 q_0 - a_3 q_3 - b_0 p_0 + b_3 p_3, \\
&\quad a_3 p_0 s_3 - a_3 p_3 s_3 - a_3 q_3 r_0 + a_3 q_3 r_3 - b_0 p_3 r_0 + b_3 p_3 r_0 + b_0 p_0 r_3 - b_3 p_0 r_3, \\
&\quad a_3 q_0 - a_3 q_3 - b_0 p_3 + b_3 p_3, a_0 s_0 - a_3 s_3 - b_0 r_0 + b_3 r_3, \\
&\quad - a_0 p_3 s_3 + a_3 p_3 s_3 + a_0 q_3 r_3 - a_3 q_3 r_3 + b_0 p_3 r_0 - b_0 p_0 r_3, \\
&\quad - a_3 b_0 r_0 + a_3 b_3 r_3 + a_0 b_0 r_3 - a_0 b_3 r_3 - a_3^2 s_3 + a_0 a_3 s_3, \\
&\quad - a_3 b_0 p_0 + a_3 b_3 p_3 + a_0 b_0 p_3 - a_0 b_3 p_3 - a_3^2 q_3 + a_0 a_3 q_3 \rangle
\end{aligned}$$

where

$$J_2 = \langle b_3 - b_2 + a_2 m, q_3 - q_2 + p_2 m, s_3 - s_2 + r_2 m, a_3 - a_2, p_3 - p_2, r_3 - r_2 \rangle$$

By continuing this computation we get the following ideals $\mathcal{I}_4$ and $\mathcal{I}_6$ which are the invariant ideals after two and three iterations of the outer loop L respectively.

$$\begin{aligned}
\mathcal{I}_4 = \langle & p_0 s_0 - p_5 s_5 - r_0 q_0 + r_5 q_5, \\
& b_5 p_5 - b_0 p_0 + a_0 q_0 - a_5 q_5, \\
& b_5 r_5 - b_0 r_0 + a_0 s_0 - a_5 s_5, \\
& b_5(-p_5 s_0 + r_5 q_0) + b_0(p_5 s_5 - r_5 q_5) + a_5(-s_5 q_0 + s_0 q_5), \\
& b_5(-p_5 r_0 + p_0 r_5) + a_5(-p_0 s_5 + r_0 q_5) + a_0(p_5 s_5 - r_5 q_5), \\
& b_0 p_0(-p_5 s_5 + r_5 q_5) + b_5(p_5^2 s_5 - p_0 r_5 q_0 + p_5(r_0 q_0 - r_5 q_5)) + \\
& \quad a_5(p_0 s_5 q_0 + q_5(-p_5 s_5 - r_0 q_0 + r_5 q_5)) \rangle
\end{aligned}$$

$$\begin{aligned}
\mathcal{I}_6 = \langle & p_0 s_0 - p_7 s_7 - r_0 q_0 + r_7 q_7, \\
& b_7 p_7 - b_0 p_0 + a_0 q_0 - a_7 q_7, \\
& b_7 r_7 - b_0 r_0 + a_0 s_0 - a_7 s_7, \\
& b_7(-p_7 s_0 + r_7 q_0) + b_0(p_7 s_7 - r_7 q_7) + a_7(-s_7 q_0 + s_0 q_7), \\
& b_7(-p_7 r_0 + p_0 r_7) + a_7(-p_0 s_7 + r_0 q_7) + a_0(p_7 s_7 - r_7 q_7), \\
& b_0 p_0(-p_7 s_7 + r_7 q_7) + b_7(p_7^2 s_7 - p_0 r_7 q_0 + p_7(r_0 q_0 - r_7 q_7)) + \\
& \quad a_7(p_0 s_7 q_0 + q_7(-p_7 s_7 - r_0 q_0 + r_7 q_7)) \rangle
\end{aligned}$$

Note that we now reached the fixed point as $\mathcal{I}_6 = \mathcal{I}_4|_{\boldsymbol{v}^{(5)} \leftarrow \boldsymbol{v}^{(7)}}$.

Corollary 22 provides a bound on the number of iterations in Algorithm 1. Therefore, we know at which stage we have to reach the fixed point of the computation at the latest, viz. after computing $\mathcal{I}_{r \cdot m}$. This fact allows us to construct a new algorithm which computes the ideal $\mathcal{I}_{r \cdot m}$ directly instead of doing a fixed point computation. The benefit of Algorithm 2 is that we have

to perform only one Gröbner basis computation in the end, although the new algorithm might performs more iterations than Algorithm 1.

Algorithm 2 Invariant generation without fixed point computation

Input: Loop bodies $B_1, \ldots, B_r$ as described.
Output: The ideal of all polynomial invariants of $(B_1^*; B_2^*; \ldots; B_r^*)^*$.

1: Compute $I := I(\boldsymbol{\theta}, \boldsymbol{\zeta})$ as described above
2: $\mathcal{I}_{new} = \sum_{j=1}^{m} \langle v_j^{(1)} - v_i^{(0)} \rangle + I$
3: **FOR** $j = 1, \ldots, m$ **DO**
4: **FOR** $i = 1, \ldots, r$ **DO**
5: $\mathcal{I}_{new} \leftarrow (J_{i \cdot j} + \mathcal{I}_{new})$
6: **RETURN** $\mathcal{I}_{new} \cap \mathbb{K}[\boldsymbol{v}^{(m \cdot r+1)}, \boldsymbol{v}^{(0)}]$

The proof of termination of the invariant generation method of [10] assumes that the ideal of algebraic dependencies is prime. In general, this does not hold. Consider the following loop and its closed forms with exponential sequences 2^n and $(-2)^n$:

while ... **do**
$x := 2x$ $\qquad x(n) = 2^n \cdot x(0)$
$y := -2y$ $\qquad y(n) = (-2)^n \cdot y(0)$
end while

The ideal of algebraic dependencies among the before-mentioned exponential sequences is given by $\langle a^2 - b^2 \rangle$ which is obviously not prime. As a consequence, the termination proof of [10] is incorrect. This paper closes this gap by providing a new algorithm and a corresponding termination proof.

4 Implementation and Experiments

We implemented our method in the Mathematica package Aligator[1]. Aligator is open source and available at:

`https://ahumenberger.github.io/aligator/`

Comparison of generated invariants. Based on the examples in Figure 1 we show that our technique can infer invariants which cannot be found by other state-of-the-art approaches. Our observations indicate that our method is superior to existing approaches if the loop under consideration has some *mathematical meaning* like division or factorization algorithms as depicted in Figure 1, whereas the approach of [7] has advantages when it comes to programs with complex flow.

[1] Aligator requires the Mathematica packages Hyper [14], Dependencies [6] and FastZeil [13], where the latter two are part of the compilation package ErgoSum [15].

The techniques of [2] and [7] were implemented in tools called FASTIND[2] and DUET[3] respectively. Unlike ALIGATOR and FASTIND, DUET is not a pure inference engine for polynomial invariants, instead it tries to prove user-specified safety assertions. In order to check which invariants can be generated by DUET, we therefore asserted the invariants computed by ALIGATOR and checked if DUET can prove them.

```
(a)
while a ≠ b do
  if a > b then
    a := a − b
    p := p − q
    r := r − s
  else
    b := b − a
    q := q − p
    s := s − r
  end if
end while

(b)
while r ≠ 0 do
  if r > 0 then
    r := r − v
    v := v + 2
  else
    r := r + u
    u := u + 2
  end if
end while

(c)
while d ≥ E do
  if P < a + b then
    b := b/2
    d := d/2
  else
    a := a + b
    y := y + d/2
    b := b/2
    d := d/2
  end if
end while
```

Fig. 1: Three examples: (a) Extended Euclidean algorithm, (b) a variant of Fermat's factorization algorithm and (c) Wensley's algorithm for real division.

Let us consider the loop depicted in Figure 1a. Since we treat conditional branches as inner loops, we have that the invariants for this loop are the same as for the loop in Example 23. By instantiating the generated invariants with the following initial values on the left we get the following polynomial invariants on the right:

$$a_0 \mapsto x \qquad 1 + qr - ps \qquad (I_1)$$
$$b_0 \mapsto y \qquad bp - aq - y \qquad (I_2)$$
$$p_0 \mapsto 1 \qquad br - as + x \qquad (I_3)$$
$$q_0 \mapsto 0 \qquad -bp + aq - qry + psy \qquad (I_4)$$
$$r_0 \mapsto 0 \qquad br - as - qrx + psx \qquad (I_5)$$
$$s_0 \mapsto 1 \qquad (qr - ps)(-bp + aq + y) \qquad (I_6)$$

Note that (I_4)-(I_6) are just linear combinations of (I_1)-(I_3). However, FASTIND was able to infer (I_1)-(I_3), whereas DUET was only able to prove (I_2), (I_5) and (I_6).

Other examples where ALIGATOR is superior in terms of the number of inferred invariants are given by the loops in Figures 1b and 1c. For Fermat's

[2] Available at http://www.irisa.fr/celtique/ext/polyinv/

[3] Available at https://github.com/zkincaid/duet

algorithm (Figure 1b) and the following initial values, ALIGATOR found one invariant, which was also found by FASTIND. However, DUET was not able to prove it.

$$u_0 \mapsto 2R+1 \qquad v_0 \mapsto 1 \qquad r_0 \mapsto RR-N \qquad -4N-4r-2u+u^2+2v-v^2 \quad (I_7)$$

In case of Wensley's algorithm (Figure 1c) ALIGATOR was able to identify the following three invariants. FASTIND inferred the first two invariants, whereas DUET could not prove any of them.

$$a_0 \mapsto 0 \qquad b_0 \mapsto Q/2 \qquad d_0 \mapsto 1 \qquad y_0 \mapsto 0$$

$$2b - dQ \quad (I_8)$$

$$ad - 2by \quad (I_9)$$

$$a - Qy \quad (I_{10})$$

Benchmarks and Evaluation. For the experimental evaluation of our approach, we used the following set of examples: (i) 18 programs taken from [2]; (ii) 4 new programs of extended P-solvable loops that were created by us. All examples are available at the repository of ALIGATOR.

Our experiments were performed on a machine with a 2.9 GHz Intel Core i5 and 16 GB LPDDR3 RAM; for each example, a timeout of 300 seconds was set. When using ALIGATOR, the Gröbner basis of the invariant ideal computed by ALIGATOR was non-empty for each example; that is, for each example we were able to find non-trivial invariants.

We evaluated ALIGATOR against FASTIND. As DUET is not a pure inference engine for polynomial invariants, we did not include it in the following evaluation. When compared to [2], we note that we do not fix the degree of the polynomial invariants to be generated. Moreover, our method is complete. That is, whenever ALIGATOR terminates, the basis of the polynomial invariant ideal is inferred; any other polynomial invariant is a linear combination of the basis polynomials.

Table 1a summarizes our experimental results on single-path loops, whereas Table 1b reports on the results from multi-path programs. The first column of each table lists the name of the benchmark. The second and third columns of Table 1a report, on the timing results of ALIGATOR and FASTIND, respectively. In Table 1b, the second column lists the number of branches (paths) of the multi-path loop, whereas the third column gives the number of variables used in the program. The fourth column reports on the number of iterations until the fixed point is reached by ALIGATOR, and hence terminates. The fifth and sixth columns, labeled AL1 and AL2, show the performance of ALIGATOR when using Algorithm 1 or Algorithm 2, respectively. The last column of Table 1b lists the results obtained by FASTIND. In both tables, timeouts are denoted by

[4] Testing the Maple implementation was not possible due to constraints regarding the Maple version.

Table 1: Experimental evaluation of Aligator.

(a)

Single-path	Aligator	FastInd
`cohencu`	0.072	0.043
`freire1`	0.016	0.041
`freire2`	0.062	0.048
`petter1`	0.015	0.040
`petter2`	0.026	0.042
`petter3`	0.035	0.051
`petter4`	0.042	0.104
`petter5`	0.053	0.261
`petter20`	48.290	9.816
`petter22`	247.820	9.882
`petter23`	*TO*	9.853

(b)

Multi-path	$\#b$	$\#v$	$\#i$	Al1	Al2	FastInd
`divbin`	2	3	2	0.134	45.948	0.045
`euclidex`	2	6	3	0.433	*TO*	0.049
`fermat`	2	3	2	0.045	0.060	0.043
`knuth`	4	5	2	55.791	*TO*	1.025
`lcm`	2	4	3	0.051	87.752	0.043
`mannadiv`	2	3	2	0.022	0.025	0.048
`wensley`	2	4	2	0.124	41.851	*err*
`extpsolv2`	2	3	2	0.192	*TO*	*err*
`extpsolv3`	3	3	2	0.295	*TO*	*err*
`extpsolv4`	4	3	2	0.365	*TO*	*err*
`extpsolv10`	10	3	2	0.951	*TO*	*err*

$\#b, \#v$ ... number of branches, variables
$\#i$ ... number of iterations until fixed point reached
Al1 ... Aligator with Algorithm 1 (timeout 300s)
Al2 ... Aligator with Algorithm 2 (timeout 100s)
FastInd ... OCaml version of the tool in [2][4]
TO, err ... timeout, error

TO, whereas errors, due to the fact that the tool cannot be evaluated on the respective example, are given as *err*.

The results reported in Tables 1a and 1b show the efficiency of Aligator: in 14 out of 18 examples, Aligator performed significantly better than FastInd. For the examples `petter20`, `petter22` and `petter23`, the time-consuming part in Aligator comes from recurrence solving (computing the closed form of the recurrence), and not from the Gröbner basis computation. We intend to improve this part of Aligator in the future. The examples `extpsolv2`, `extpsolv3`, `extpsolv4` and `extpsolv10` are extended P-solvable loops with respectively 2, 3, 4, and 10 nested conditional branches. The polynomial arithmetic of these examples is not supported by FastInd. The results of Aligator on these examples indicate that extended P-solvable loops do not increase the complexity of computing the invariant ideal.

We also compared the performance of Aligator with Algorithm 1 against Algorithm 2. As shown in columns 5 and 6 of Table 1b, Algorithm 2 is not as efficient as Algorithm 1, even though Algorithm 2 uses only a single Gröbner basis computation. We conjecture that this is due to the increased number of variables in the polynomial system which influences the Gröbner basis computation. We therefore conclude that several small Gröbner basis computations (with fewer variables) perform better than a single large one.

5 Conclusions

We proposed a new algorithm for computing the ideal of all polynomial invariants for the class of extended P-solvable multi-path loops. The new approach computes the invariant ideal for a non-deterministic program $(L_1; \ldots ; L_r)^*$ where the L_i are single-path loops. As a consequence, the proposed method can handle loops containing (i) an arbitrary nesting of conditionals, as these conditional branches can be transformed into a sequence of single-path loops by introducing flags, and (ii) one level of nested single-path loops.

Our method computes the ideals $\mathcal{I}_1, \mathcal{I}_2, \ldots$ until a fixed point is reached where $\mathcal{I}_i$ denotes the invariant ideal of $(L_1; \ldots ; L_r)^i$. This fixed point is then a basis for the ideal containing all polynomial invariants for the extended P-solvable loop. We showed that this fixed point computation is guaranteed to terminate which implies the completeness of our method. Furthermore, we gave a bound on the number of iterations we have to perform to reach the fixed point. The proven bound is given by m iterations where m is the number of loop variables.

We showed that our method can generate invariants which cannot be inferred by other state-of-the-art techniques. In addition, we showcased the efficiency of our approach by comparing our Mathematica package Aligator with state-of-the-art tools in invariant generation.

Future research directions include the incorporation of the loop condition into our method. So far we operate on an abstraction of the loop where we ignore the loop condition and treat the loop as a non-deterministic program. By doing so we might loose valuable information about the control flow of the program. By employing $\Pi\Sigma^*$-theory [19] it might be possible to extend our work also to loops containing arbitrary nesting of inner loops, which reflects another focus for further research.

Acknowledgments. We want to thank the anonymous reviewers for their helpful comments and remarks.

References

1. Buchberger, B.: An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. J. Symbolic Computation 41(3-4), 475–511 (2006)
2. Cachera, D., Jensen, T.P., Jobin, A., Kirchner, F.: Inference of Polynomial Invariants for Imperative Programs: A Farewell to Gröbner Bases. In: Miné, A., Schmidt, D. (eds.) Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7460, pp. 58–74. Springer (2012)
3. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: Proc. of FMCAD. pp. 57–64. FMCAD Inc, Austin, TX (2015)
4. Humenberger, A., Jaroschek, M., Kovács, L.: Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. In: Proceedings of the 2017

ACM on International Symposium on Symbolic and Algebraic Computation. pp. 221–228. ISSAC '17, ACM, New York, NY, USA (2017)
5. Kauers, M., Paule, P.: The Concrete Tetrahedron. Text and Monographs in Symbolic Computation, Springer Wien, 1st edn. (2011)
6. Kauers, M., Zimmermann, B.: Computing the algebraic relations of C-finite sequences and multisequences. Journal of Symbolic Computation 43(11), 787 – 803 (2008)
7. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-Linear Reasoning For Invariant Synthesis. In: POPL (2018), to appear
8. Kovács, L.: Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema. Ph.D. thesis, RISC, Johannes Kepler University Linz (October 2007)
9. Kovács, L.: Reasoning Algebraically About P-Solvable Loops. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 249–264 (2008)
10. Kovács, L.: A Complete Invariant Generation Approach for P-solvable Loops. In: Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers. pp. 242–256 (2009)
11. Müller-Olm, M., Seidl, H.: A Note on Karr's Algorithm. In: Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings. pp. 1016–1028 (2004)
12. de Oliveira, S., Bensalem, S., Prevosto, V.: Polynomial invariants by linear algebra. In: Artho, C., Legay, A., Peled, D. (eds.) Proc. of ATVA. pp. 479–494. Springer (2016)
13. Paule, P., Schorn, M.: A Mathematica Version of Zeilbergers Algorithm for Proving Binomial Coefficient Identities. Journal of Symbolic Computation 20, 673 – 698 (1995)
14. Petkovšek, M.: Mathematic package hyper (1998), `http://www.fmf.uni-lj.si/~petkovsek/`
15. Research Institute for Symbolic Computation.: Mathematic Package ErgoSum (2016), `http://www.risc.jku.at/research/combinat/software/ergosum/`
16. Rodriguez-Carbonell, E., Kapur, D.: Automatic Generation of Polynomial Invariants of Bounded Degree using Abstract Interpretation. J. Science of Computer Programming 64(1), 54–75 (2007)
17. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. Journal of Symbolic Computation 42(4), 443 – 476 (2007)
18. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using gröbner bases. In: Proc. of POPL. pp. 318–329. ACM, New York, NY, USA (2004)
19. Schneider, C.: Summation theory ii: Characterizations of $r\pi\sigma$-extensions and algorithmic aspects. J. Symb. Comput. 80(3), 616–664 (2017), arXiv:1603.04285 [cs.SC]
20. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A Data Driven Approach for Algebraic Loop Invariants. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 574–592. Springer (2013)

Analyzing Guarded Protocols: Better Cutoffs, More Systems, More Expressivity

Swen Jacobs, Mouhammad Sakr

Reactive Systems Group, Saarland University, Germany
{jacobs,sakr}@react.uni-saarland.de

Abstract. We study cutoff results for parameterized verification and synthesis of guarded protocols, as introduced by Emerson and Kahlon (2000). Guarded protocols describe systems of processes whose transitions are enabled or disabled depending on the existence of other processes in certain local states. Cutoff results reduce reasoning about systems with an arbitrary number of processes to systems of a determined, fixed size. Our work is based on the observation that existing cutoff results for guarded protocols are often impractical, since they scale linearly in the number of local states of processes in the system. We provide new cutoffs that scale not with the number of local states, but with the number of guards in the system, which is in many cases much smaller. Furthermore, we consider generalizations of the type of guards and of the specifications under consideration, and present results for problems that have not been known to admit cutoffs before.

1 Introduction

Concurrent systems are notoriously hard to get correct, and are therefore a promising application area for formal methods like model checking or synthesis. However, while such general-purpose formal methods can give strong correctness guarantees, they have two drawbacks: i) the state explosion problem prevents us from using them for systems with a large number of components, and ii) correctness properties are often expected to hold for an *arbitrary* number of components, which cannot be guaranteed without an additional argument that extends a proof of correctness to systems of arbitrary size. Both problems can be solved by approaches for *parameterized* model checking and synthesis, which give correctness guarantees for systems with any number of components without considering every possible system instance explicitly.

While the parameterized model checking problem (PMCP) is undecidable even if we restrict systems to uniform finite-state components [25], there exist several methods that decide the problem for specific classes of systems [2–4, 11, 15–17, 19, 21], many of which have been collected in surveys of the literature recently [9, 20]. Additionally, there are semi-decision procedures that are successful in a number of interesting cases [1, 10, 12, 23, 24]. In this paper, we consider the *cutoff* approach to the PMCP, that can guarantee properties of systems of arbitrary size by considering only systems of up to a certain fixed size. Thus, it

© Springer International Publishing AG 2018

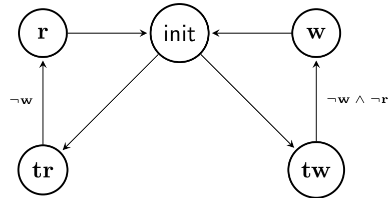
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 247–268, 2018.
https://doi.org/10.1007/978-3-319-73721-8_12

provides a decision procedure for the PMCP if the model checking problem for a fixed number of components is decidable, e.g., if components are finite-state.

Guarded protocols, the systems under consideration, are composed of an arbitrary number of processes, each an instance of a finite-state process template. The process templates can be seen as synchronization skeletons [14], i.e., they only have to model the features of the system components that are important for their synchronization. Processes communicate by guarded updates, where guards are statements about other processes that are interpreted either conjunctively ("every other process satisfies the guard") or disjunctively ("there exists a process that satisfies the guard"). Conjunctive guards can be used to model synchronization by atomic sections or locks, while disjunctive guards can model pairwise rendezvous or token-passing.

Emerson and Namjoshi [18] have shown that the PMCP for systems that combine conjunctive and disjunctive guards is in general undecidable. Therefore, research in the literature has focused on systems that are restricted to one type of guard, called conjunctive or disjunctive systems, respectively. These classes of systems have been studied by Emerson and Kahlon [15,16], and cutoffs that depend on the size of process templates are known for specifications of the form $\forall \bar{p}.\ \Phi(\bar{p})$, where $\Phi(\bar{p})$ is an $\mathsf{LTL}\backslash\mathsf{X}$ property over the local states of one or more processes $\bar{p}$. Außerlechner et al. [6] have extended and improved these results, but a number of open issues remain. We will explain some of them in the following.

Motivating Example As an example, consider the reader-writer protocol on the right, modeling access to data shared between processes. A process that wants to read the data enters state tr ("try-read"). From tr, it can move to the reading state r. However, this transition is guarded by a statement $\neg w$. Formally, guards are sets of states, and $\neg w$ stands for the set of all states except w. Furthermore, this example is a conjunctive system, which means that a guard is interpreted as "all other processes have to be in the given set of states". Thus, to take the transition from tr to r, no other process should currently be in state w, i.e., writing the data. Similarly, a process that wants to enter w has to go through tw, but the transition into w is only enabled if no state is reading or writing.



For this example, consider parameterized safety conditions such as

$$\forall i \neq j.\, \mathsf{G}\,(\neg(w_i \wedge w_j) \wedge \neg(w_i \wedge r_j))\,,$$

where indices i and j refer to processes in the system. Emerson and Kahlon [15] show that properties from $\mathsf{LTL}\backslash\mathsf{X}$ of a single process have a cutoff of 2, which generalizes to a cutoff of $k+1$ for properties with k index variables. Moreover, they show that a cutoff linear in the size of the process template is sufficient to detect global deadlocks.

However, Außerlechner et al. [6] noted that for liveness properties such as

$$\forall i.\, \mathsf{G}\,((tr_i \rightarrow \mathsf{F}\, r_i) \wedge (tw_i \rightarrow \mathsf{F}\, w_i))\,,$$

an explicit treatment of fairness assumptions on the scheduling of processes is necessary. They show that the cutoff for $\mathsf{LTL}\backslash\mathsf{X}$ properties also holds under fairness assumptions, but a second aspect has to be considered to adequately treat liveness properties in guarded protocols: the detection of local deadlocks, i.e., whether some process stops after finite time. For this problem, they give a cutoff that is linear in the size of the process template, but a major restriction is that the cutoff only supports systems with 1-conjunctive guards, i.e., where each guard can only exclude a single state. Note that the example above is not supported by these results, since one of the guards excludes 2 states.

Another drawback of the existing results is that they use only minimal knowledge about the process templates: their size and the interpretation of guards. As a result, many cutoffs depend directly on the size of the process template. Intuitively, the *communication* between processes should be more important for the cutoff than their internal state space. This can also be seen in the example above: out of the 5 states, only 2 can be observed by the other processes, and can thus influence their behavior. In this paper, we will explore the idea of cutoffs that depend on the number and form of guards in the system.

Contributions We provide new cutoff results for guarded protocols:

1. For conjunctive systems, we extend the class of process templates that are supported by cutoff results, providing cutoff results for local deadlock detection in classes of templates that are not 1-conjunctive, and include examples such as the one above. However, we do not solve the general problem, and instead show that a cutoff for arbitrary conjunctive systems has to be at least quadratic in the size of the template.
2. For both conjunctive and disjunctive systems, we show that by closer analysis of process templates, in particular the number and the form of transition guards, we can obtain smaller cutoffs in many cases. This circumvents the tightness results of Außerlechner et al. [6], which state that no smaller cutoffs can exist for the class of all processes of a given size.
3. For disjunctive systems, we additionally extend both the class of process templates and the class of specifications that are supported by cutoff results. We show that systems with finite conjunctions of disjunctive guards are also supported by variations of the existing proof methods, and obtain cutoff results for these systems. Furthermore, we give cutoffs that support checking the simultaneous reachability (and repeated reachability) of a target set by *all* processes in a disjunctive system.

2 Preliminaries

2.1 System Model

In the following, let Q be a finite set of states.

Processes. A *process template* is a transition system $U = (Q_U, \mathsf{init}_U, \delta_U)$ with[1]

- $Q_U \subseteq Q$ is a finite set of states including the initial state init_U,
- $\delta_U : Q_U \times \mathcal{P}(Q) \times Q_U$ is a guarded transition relation.

Define the size of U as $|U| = |Q_U|$. An instance of template U will be called a *U-process*.

Guarded Protocols. Fix process templates A and B. A *guarded protocol* is a system $A \| B^n$, consisting of one A-process and n B-processes in an interleaving parallel composition.[2] We assume that $Q = Q_A \dot\cup Q_B$. Different B-processes are distinguished by subscript, i.e., for $i \in [1..n]$, B_i is the ith instance of B, and q_{B_i} is a state of B_i. A state of the A-process is denoted by q_A. We denote the set $\{A, B_1, \ldots, B_n\}$ as $\mathcal{P}$, and write p for a process in $\mathcal{P}$. For $U \in \{A, B\}$, we write G_U for the set of non-trivial guards that are used in δ_U, i.e., guards different from Q and $\emptyset$. Then, let $G = G_A \cup G_B$.

Disjunctive and Conjunctive Systems. In a guarded protocols, a local transition $(q_p, g, q'_p) \in \delta_U$ of p is *enabled in* s if its *guard* g is satisfied for p in s, written $(s, p) \models g$. We distinguish two types of guarded protocols, depending on their *interpretation of guards*:

$$\text{In disjunctive systems: } (s,p) \models g \quad \text{iff} \quad \exists p' \in \mathcal{P} \setminus \{p\} : \quad q_{p'} \in g.$$
$$\text{In conjunctive systems: } (s,p) \models g \quad \text{iff} \quad \forall p' \in \mathcal{P} \setminus \{p\} : \quad q_{p'} \in g.$$

Let $\mathsf{set}(s) = \{q_A, q_{B_1}, \ldots, q_{B_n}\}$, and for a set of processes $P = \{p_1, \ldots, p_k\}$, let $\mathsf{set}_P(s) = \{q_{p_1}, \ldots, q_{p_k}\}$. Then for disjunctive systems, we can more succinctly state that $(s,p) \models g$ iff $\mathsf{set}_{\mathcal{P} \setminus p}(s) \cap g \neq \emptyset$, and for conjunctive systems $(s,p) \models g$ iff $\mathsf{set}_{\mathcal{P} \setminus p}(s) \subseteq g$. A process is *enabled* in s if at least one of its transitions is enabled in s, otherwise it is *disabled*.

Like Emerson and Kahlon [15], we assume that in conjunctive systems init_A and init_B are contained in all guards, i.e., they act as neutral states. For conjunctive systems, we call a guard *k-conjunctive* if it is of the form $Q \setminus \{q_1, \ldots, q_k\}$ for some $q_1, \ldots, q_k \in Q$. A state q is *k-conjunctive* if all non-trivial guards of transitions from q are k'-conjunctive with $k' \leq k$. A conjunctive system is *k-conjunctive* if every state is k-conjunctive.

Then, $A \| B^n$ is defined as the transition system $(S, \mathsf{init}_S, \Delta)$ with

- set of global states $S = (Q_A) \times (Q_B)^n$,
- global initial state $\mathsf{init}_S = (\mathsf{init}_A, \mathsf{init}_B, \ldots, \mathsf{init}_B)$,
- and global transition relation $\Delta \subseteq S \times S$ with $(s, s') \in \Delta$ iff s' is obtained from $s = (q_A, q_{B_1}, \ldots, q_{B_n})$ by replacing one local state q_p with a new local state q'_p, where p is a U-process with local transition $(q_p, g, q'_p) \in \delta_U$ and $(s, p) \models g$.

[1] In contrast to Außerlechner et al. [6], for simplicity we only consider closed process templates. However, our results extend to open process templates in the same way as explained there.

[2] By similar arguments as in Emerson and Kahlon [15], our results can be extended to systems with an arbitrary number of process templates.

Runs. A *path* of a system is a sequence of states $x = s_1, s_2, \ldots$ such that for all $m < |x|$ there is a transition $(s_m, s_{m+1}) \in \Delta$ based on a local transition of some process p_m. We say that process p_m *moves* at *moment* m. A path can be finite or infinite, and a *maximal path* is a path that cannot be extended, i.e., it is either infinite or ends in a state where no transition is enabled.

A system *run* is a maximal path starting in the initial state. We say that a run is *initializing* if every process that moves infinitely often also visits its initial state init infinitely often.

Given a system path $x = s_1, s_2, \ldots$ and a process p, the *local path* of p in x is the projection $x(p) = s_1(p), s_2(p), \ldots$ of x onto local states of p. A local path $x(p)$ is a *local run* if x is a run.

Deadlocks and Fairness. A run is *globally deadlocked* if it is finite. An infinite run is *locally deadlocked* for process p if there exists m such that p is disabled for all $s_{m'}$ with $m' \geq m$. A run is *deadlocked* if it is locally or globally deadlocked. A system *has a (local/global) deadlock* if it has a (locally/globally) deadlocked run. Note that absence of local deadlocks for all p implies absence of global deadlocks, but not the other way around.

A run $s_1, s_2, \ldots$ is *unconditionally fair* if every process moves infinitely often. A run is *strong fair* if it is infinite and for every process p, if p is enabled infinitely often, then p moves infinitely often.

Remark. We consider these different notions of fairness for the following reason: we are interested in unconditionally fair runs of the system, which requires an assumption about scheduling. However, directly assuming unconditional fairness is too strong, since any run with a local deadlock will violate the assumption, and therefore satisfy the overall specification. Thus, we consider strong fairness as an assumption on the scheduler, and absence of local deadlocks as a property of the system that has to be proved. Together, they imply unconditional fairness.

2.2 Specifications

We consider formulas in $\mathsf{LTL}\backslash\mathsf{X}$, i.e., LTL without the next-time operator X. Let $h(A, B_{i_1}, \ldots, B_{i_k})$ be an $\mathsf{LTL}\backslash\mathsf{X}$ formula over atomic propositions from Q_A and indexed propositions from $Q_B \times \{i_1, \ldots, i_k\}$. For a system $A \| B^n$ with $n \geq k$ and $i_j \in [1..n]$, satisfaction of $\mathsf{A}\, h(A, B_{i_1}, \ldots, B_{i_k})$ and $\mathsf{E}\, h(A, B_{i_1}, \ldots, B_{i_k})$ is defined in the usual way (see e.g. Baier and Katoen [7]).

Parameterized Specifications. A *parameterized specification* is a temporal logic formula with indexed atomic propositions and quantification over indices. A *k-indexed formula* is of the form $\forall i_1, \ldots, i_k.\, \mathsf{A}\, h(A, B_{i_1}, \ldots, B_{i_k})$ or $\forall i_1, \ldots, i_k.\, \mathsf{E}\, h(A, B_{i_1}, \ldots, B_{i_k})$. For given $n \geq k$, by symmetry of guarded protocols (cp. Emerson and Kahlon [15]) we have

$$A \| B^n \models \forall i_1, \ldots, i_k.\, \mathsf{A}\, h(A, B_{i_1}, \ldots, B_{i_k}) \quad \text{iff} \quad A \| B^n \models \mathsf{A}\, h(A, B_1, \ldots, B_k).$$

The latter formula is denoted by $\mathsf{A}\, h(A, B^{(k)})$, and we often use it instead of the original $\forall i_1, \ldots, i_k.\, \mathsf{A}\, h(A, B_{i_1}, ..., B_{i_k})$. For formulas with path quantifier E, satisfaction is defined analogously, and equivalent to satisfaction of $\mathsf{E}\, h(A, B^{(k)})$.

2.3 Model Checking Problems and Cutoffs

For a given system $A\|B^n$ and specification $h(A, B^{(k)})$ with $n \geq k$,

- the *model checking problem* is to decide whether $A\|B^n \models \mathsf{A}\, h(A, B^{(k)})$,
- the (global/local) *deadlock detection problem* is to decide whether $A\|B^n$ has (global/local) deadlocks,
- the *parameterized model checking problem* (PMCP) is to decide whether $\forall m \geq n : \ A\|B^m \models \mathsf{A}\, h(A, B^{(k)})$, and
- the *parameterized (local/global) deadlock detection problem* is to decide whether for some $m \geq n$, $A\|B^m$ does have (global/local) deadlocks.

These definitions can be flavored with different notions of fairness, and with the E path quantifier instead of A. According to our remarks about fairness above, we are interested in proving the absence of local deadlocks under the assumption of strong fairness, which implies unconditional fairness and therefore allows us to separately prove the satisfaction of a temporal logic specification under the assumption of unconditional fairness.

Corresponding problems for the *synthesis* of process templates can be defined (compare Außerlechner et al. [6]). Parameterized synthesis based on cutoffs [22] is also supported by our cutoff results, but the details will not be necessary for understanding the results presented here.

Cutoffs. We define cutoffs with respect to a class of systems (either disjunctive or conjunctive), a class of process templates T, and a class of properties, which can be k-indexed formulas for some $k \in \mathbb{N}$ or the existence of (local/global) deadlocks. A *cutoff* for a given class of properties and a class of systems with processes from T is a number $c \in \mathbb{N}$ such that for all $A, B \in T$, all properties φ in the given class, and all $n \geq c$:

$$A\|B^n \models \varphi \ \Leftrightarrow \ A\|B^c \models \varphi.$$

Like the problem definitions above, cutoffs may additionally be flavoured with different notions of fairness.

Cutoffs and Decidability. Note that the existence of a cutoff implies that the parameterized model checking and parameterized deadlock detection problems are *decidable* iff their non-parameterized versions are decidable.

3 New Cutoff Results for Conjunctive Systems

In this section, we state our new results for conjunctive systems, and compare them to the previously known results in Table 1. We give improved cutoffs for global deadlock detection in general (Section 3.1), and for local deadlock detection for the restricted case of 1-conjunctive systems (Section 3.2). After that, we explain why local deadlock detection in general is hard, and identify a number of cases where we can solve the problem even for systems that are not 1-conjunctive (Sections 3.3 and 3.4). We do not improve on the cutoffs for $\mathsf{LTL}\backslash\mathsf{X}$ properties, since they are already very small and only depend on the number of index variables in the specification.

Additional Definitions. To analyze deadlocks in a given conjunctive system $A \| B^n$, we introduce additional definitions. A *deadset* is a minimal set D of local states of other processes that block all outgoing transitions of one process in its current state q. Formally, we say that $D \subseteq Q$ is a *deadset* of $q \in Q$ if:

i) $\forall (q, g, q') \in \delta : \exists q'' \in D : q'' \notin g$,
ii) D contains at most one state from Q_A, and
iii) there is no D' that satisfies i) and ii) with $D' \subset D$.

For a given local state q, $dead_q^\wedge$ is the set of all deadsets of q:

$$dead_q^\wedge = \{D \subseteq Q \mid D \text{ is a deadset of } q\}.$$

If $dead_q^\wedge = \emptyset$, then we say q is *free*. If a state q does not appear in $dead_{q'}^\wedge$ for any $q' \in Q$, then we say q is *non-blocking*. If a state q does not appear in $dead_q^\wedge$, then we say q is *not self-blocking*.

For example, in a system where B is the process template from Sect. 1, we have $dead_{tw}^\wedge = \{\{w\}, \{r\}\}$ and $dead_{tr}^\wedge = \{\{w\}\}$. For all other states $q \in \{\mathsf{init}, r, w\}$, we have $dead_q^\wedge = \emptyset$. Regardless of A, none of the deadsets contain a state from A, since the guards of B do not mention states of A.

In these terms, a globally deadlocked run is a run that ends in a global state s such that for every process p and its local state q, some $D \in dead_q^\wedge$ is contained in $\mathsf{set}(s)$. Similarly, a locally deadlocked run is a run such that one process p will eventually always remain in state q, and from some point on, we always have $D \subseteq \mathsf{set}(s)$ for some $D \in dead_q^\wedge$. Note that in this case, it can happen that there does not exist a single deadset D that is contained in $\mathsf{set}(s)$ all the time, but the run may *alternate* between different deadsets of q that are contained in $\mathsf{set}(s)$ at different times.

3.1 Global Deadlock Detection

For global deadlock detection, we show how to obtain improved cutoffs based on the number of free, non-blocking, and not self-blocking states in a given process template.

Theorem 1. *For conjunctive systems and process templates A, B, let*

- $k_1 = |D_1|$, *where $D_1 \subseteq Q_B$ is the set of free states in B,*
- $k_2 = |D_2 \setminus D_1|$, *where $D_2 \subseteq Q_B$ is the set of non-blocking states in B, and*
- $k_3 = |D_3 \setminus (D_1 \cup D_2)|$, *where $D_3 \subseteq Q_B$ is the set of not self-blocking states in B.*

Then $2|B| - 2k_1 - 2k_2 - k_3$ is a cutoff for global deadlock detection.

Proof Sketch. In order to simulate a globally deadlocked run $x = s_0, s_1, \ldots, s_m$ of a large system by a run y in the cutoff system, by Emerson and Kahlon [15] the following is sufficient. We analyze the set of local states $q \in Q$ that are present in the final state s_m of x, and distinguish whether any such q appears once in s_m, or multiple times. If q appears once, we identify one local run of x that ends in q, and replicate it in the cutoff system. If q appears multiple times, we do the same for two local runs of x that end in q. This construction ensures that in the resulting global run $x' = s'_0, \ldots, s'_m$ of the cutoff system, for any point in time t and any process p, we have $\mathsf{set}_p(s'_t) \subseteq \mathsf{set}_p(s_t)$. Therefore, all transitions in x' will be enabled, and x' is deadlocked in s'_m. If x' does not contain all local runs of x then there are stuttering steps in x', where no process moves. By removing these stuttering steps, we obtain the desired run y.

The construction of Emerson and Kahlon assumes that in the worst case all local states of B appear in the deadlocked state s_m. However, if $D_1 \subseteq Q_B$ are *free* local states, then we know that no state from D_1 can ever appear in s_m, and thus the cutoff is reduced by $2|D_1|$. Similarly, if $D_2 \subseteq Q_B$ are *non-blocking* states, then we know that no state from D_2 can be necessary for the deadlock in s_m, and therefore the construction will also work if we remove the local runs ending in D_2. This also reduces the cutoff by $2|D_2|$. Moreover, the original construction assumes that all local states q may be self-blocking, which requires the second local run that ends in q. If we know that $D_3 \subseteq Q_B$ are *not self-blocking*, then we only need one local run for each of these states, reducing the cutoff by $|D_3|$. If we combine all three cases, we get the statement of the theorem. ☐

Note that the sets of free, non-blocking, and not self-blocking states can be identified by a simple analysis of a single process template, and the cost of this analysis is negligible compared to the cost of a higher cutoff in verification of the system.

3.2 Local Deadlock Detection in 1-conjunctive Systems

For local deadlock detection, we first show that smaller cutoffs can be found by taking into account the transitions and guards of the process template. For a 1-conjunctive process template $U \in \{A, B\}$, let $G_{U,B}$ be the set of guards of U that exclude one of the states of B, i.e., that are of the form $g = Q \setminus \{q\}$ for some $q \in Q_B$. Furthermore, let $\mathsf{maxD}_U = max\{|D \cap Q_B| \mid D \in dead^{\wedge}_q \text{ for some } q \in Q_U\}$ be the maximal number of states from B that appear in any deadset of a state in U.

Theorem 2. *For conjunctive systems with process templates A, B, if process template $U \in \{A, B\}$ is 1-conjunctive, then the following are cutoffs for local deadlock detection in a U-process in non-fair runs:*

- $\mathsf{maxD}_U + 2$, *and*
- $|G_{U,B}| + 2$.

Proof Sketch. In order to simulate a locally deadlocked run $x = s_0, s_1, \ldots$ of a large system by a run y in the cutoff system, the following construction has been presented by Außerlechner et al. [5] for non-fair runs. Suppose process p is locally deadlocked in local state q after the system has entered state s_m. We first copy the local runs of A and p. Since the system is 1-conjunctive, every local state has a unique deadset. For each q' in the deadset D of q, we copy a local run from x that is in q' at time m, and modify it such that it stays in q' forever after this point in time. Thus, the process in q is locally deadlocked because all states in D will be present at any time after m. Finally, we copy one additional local run of a process that moves infinitely often in x. As in the proof of Theorem 1, all transitions of the resulting global run x' will be enabled, and we can obtain the desired run y by de-stuttering.

Note that the original proof uses one process for every state in the unique deadset D of the deadlocked local state q, and assumes that in the worst case we have $D \supseteq Q_B$, resulting in the cutoff of $|Q_B| + 2$ for all process templates with a given set of states Q_B. However, if we take into account the guards of transitions and the individual deadsets, we can obtain smaller cutoffs: in particular, instead of assuming that the size of some deadset is $|Q_B|$, we can compute the maximal size of actual deadsets maxD_U, and replace $|Q_B|$ by maxD_U to obtain a cutoff of $\mathsf{maxD}_U + 2$. Further, note that (since the system is 1-conjunctive) maxD_U is bounded by $|G_{U,B}|$, so $|G_{U,B}| + 2$ also is a cutoff. □

Theorem 3. *For conjunctive systems and process templates A, B, if process template $U \in \{A, B\}$ is 1-conjunctive, then $2|G_{U,B}|$ is a cutoff for local deadlock detection in a U-process in strong-fair runs.*

Proof Sketch. For fair runs, the construction by Außerlechner et al. [5] is similar as in the previous proof, but additionally we need to ensure that all processes either move infinitely often or are locally deadlocked. We explain the original construction in a new way that highlights our insight.

First, identify all states $q' \in Q_B$ in the deadset D of q such that there exists a locally deadlocked local run in x that eventually stays in q'. For each of these states, copy this local run. To ensure that these local runs are locally deadlocked also in the constructed run, add the states in their deadsets to D, and if q' is self-blocking then also copy another local run from x that eventually visits the state q' and stays there. Then repeat the procedure until no more states are added to D. Note that only states that are excluded in one of the (1-conjunctive) guards can be added to D, and for each state we have copied up to two local runs from x. Thus, the size of D is bounded by $|G_{U,B}|$, and in the worst case we have added $2|G_{U,B}|$ processes until now.

Then, let $D' \subseteq D$ be the set of states for which no process has been added thus far, and let m' be the time when all local runs that have been added until now are locally deadlocked. Copy for each of the states $q' \in D'$ one local run from x that is in q' at time m', and add a process that stays in init_B until time m'. Then after moment m' we can let all processes that are in D' move in the following way: (i) choose some $q' \in D'$ (ii) let the process that is in init_B move

to q' (iii) let the process that was waiting in q' move to init_B (iv) repeat with fair choices of $q' \in D'$. Since each of these states must appear in x at any time after m' without a process being locally deadlocked in the state, there must be a local path from this state to itself in one of the local runs in x. Since for fair conjunctive systems we assume that they are initializing, this path must go through init_B, and the construction is guaranteed to work.

Note that overall, for each state in D we have copied either one or two local runs from x, so the bound for the number of these processes is still $2|G_{U,B}|$. Also note that the additional process that waits in init_U is only needed if at least one of the other processes is not locally deadlocked, thus it does not increase the needed number of processes. Finally, for the original locally deadlocked process we can distinguish two cases: i) if we have added $2|G_{U,B}|$ processes thus far, then the original process is deadlocked in a state that does not block any transition, and we can remove it since the run will exhibit a local deadlock regardless, or ii) if this is not the case, then even with the original process we need at most $2|G_{U,B}|$ processes overall. □

Note that in a 1-conjunctive process template U, we have $|G_{U,B}| \leq |Q_B| - 1$. Thus, our new cutoffs are always smaller or equal to the known cutoff from Außerlechner et al. [6].

Table 1: Cutoff Results for Conjunctive Systems

		EK [15]	AJK [6]	our work
k-indexed LTL\X	non-fair	$k+1$	$k+1$	unchanged
k-indexed LTL\X	fair	-	$k+1$	unchanged
Local Deadlock	non-fair	-	$\lvert B\rvert + 1^*$	$\mathsf{maxD}_U + 2$ and $\lvert G_{U,B}\rvert + 2^*$
Local Deadlock	fair	-	$2\lvert B\rvert - 2^{**}$	$2\lvert G_{U,B}\rvert^{**}$
Global Deadlock		$2\lvert B\rvert + 1$	$2\lvert B\rvert - 2$	$2\lvert B\rvert - 2k_1 - 2k_2 - k_3$

* : systems need to have alternation-bounded local deadlocks (see Sect. 3.4)
** : systems need to be initializing and have alternation-bounded local deadlocks
k_1: number of free states
k_2: number of non-blocking states that are not free
k_3: number of not self-blocking states that are not free or non-blocking

3.3 Local Deadlock Detection: Beyond 1-conjunctive Systems

While Theorems 2 and 3 improve on the local deadlock detection cutoff for conjunctive systems in some cases, the results are still restricted to 1-conjunctive process templates. The reason for this restriction is that when going beyond 1-conjunctive systems, the local deadlock detection cutoff (even without considering fairness) can be shown to grow at least quadratically in the number of states or guards, and it becomes very hard to determine a cutoff.

To analyze these cases, define the following: given a process template $U \in \{A, B\}$, a sequence of local states $q_1, \ldots, q_k$ is *connected* if $\forall q_i \in \{q_1, \ldots, q_k\} : \exists (q_i, g_i, q_{i+1}) \in \delta_U$. A *cycle* is a connected sequence of states $q, q_1, \ldots, q_k, q$ such that $\forall q_i, q_j \in \{q_1, \ldots, q_k\} : q_i \neq q_j$. We denote such a cycle by C_q. By abuse of notation, C_q is also used for the set of states on C_q. We denote the set of guards of the transitions on C_q as G_{C_q}.

Example 1. If we consider the process template in Figure 1 without the dashed parts, then it exhibits a local deadlock in state q_0 for 9 processes, but not for 8 processes: one process has to move to q_0, which has four deadsets: $\{a, c\}$, $\{a, d\}$, $\{b, c\}$, and $\{b, d\}$. To preserve a deadlock in q_0, the processes need to alternate between different deadsets while always at least covering one of them. To achieve this, for each cycle that starts and ends in states a, b, c, d, we need 2 processes that move along the cycle to keep all guards of q_0 covered at all times. Intuitively, one process per cycle has to be in the state of interest, or ready to enter it, and the other process is traveling on the cycle, waiting until the guards are satisfied.

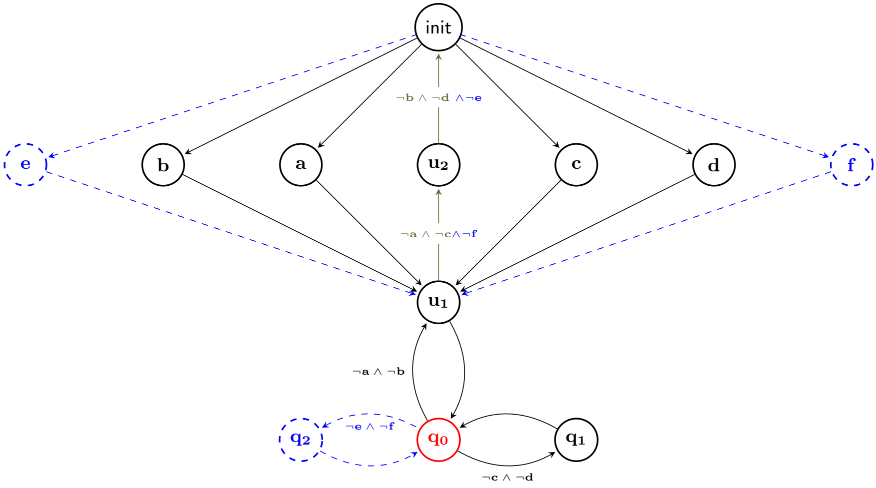


Fig. 1: Process Template with Quadratic Cutoff for Local Deadlocks

Now, consider the modified template (including the dashed parts) where we i) add two states e, f in a similar way as a, b, c, d, ii) add a new state connected to q_0 with guard $\neg e \wedge \neg f$, and iii) change the guards in the sequence from u_1 to init to $\neg a \wedge \neg c \wedge \neg e$ and $\neg b \wedge \neg d \wedge \neg f$, respectively. Then we have 6 cycles that need 2 processes each, and we need 13 processes to reach a local deadlock in q_0.

Moreover, consider the modified template where we increase the length of the path from u_1 to init by adding states u_3 and u_4, such that we obtain a sequence $(u_1, u_2, u_3, u_4, \text{init})$, where transitions alternate between the two guards from the original sequence. Then, for every cycle we need 3 processes instead

of 2, as otherwise they cannot traverse the cycle fast enough to ensure that the local deadlock is preserved infinitely long. That is, the template with both modifications now needs 19 processes to reach a local deadlock.

Observe that by increasing the height of the template, we increase the necessary number of processes without increasing the number of different guards. Moreover, when increasing both the width and height of the template, the number of processes that are necessary for a local deadlock increases quadratically with the size of the template.

This example leads us to the following result.

Theorem 4. *A cutoff for local deadlock detection for the class of all conjunctive systems must grow at least quadratically in the number of states. Furthermore, it cannot be bounded by the number of guards at all.*

Proof Sketch. For a system that does exhibit a local deadlock for some size n, but not for $n-1$, the cutoff cannot be smaller than n. Thus, the example shows that a cutoff for local deadlock detection in general is independent of the number of guards, and must grow at least quadratic in the size of the template. □

Cutoffs that can in the best case be bounded by $|B|^2$ will not be very useful in practice. Therefore, instead of solving the general problem, we identify in the following a number of cases where the cutoff remains linear in the number of states or guards.

3.4 Systems with Alternation-bounded Local Deadlocks

When comparing the proof of Theorem 2 to Example 1, we note that the reason that the cutoff in Theorem 2 does not apply is the following: while in 1-conjunctive systems every state has a unique deadset, in the general case a state may have many deadsets, and the structure of the process template may require infinitely many alternations between different deadsets to preserve the local deadlock. Moreover, as shown in the example, the number of processes needed to alternate between deadsets may increase with the size of the template, even if the set of guards (and thus, the number of different deadsets) remains the same.

However, we can still obtain small cutoffs in some cases, based on the following observation: even if states have multiple deadsets, an infinite alternation between them may not be necessary to obtain a local deadlock. In the following, we will first show that for systems where infinite alternation between different deadsets is not necessary, the cutoff for 1-conjunctive systems applies, and then give a number of sufficient conditions to identify such systems.

Alternation-bounded Local Deadlocks. We say that a run $x = s_0, s_1, \ldots$ where process p is locally deadlocked in state q *is alternation-bounded* if there is a moment m and a single set $D \subseteq Q_B$ such that for all $m' > m$: $D \subseteq \mathsf{set}_{\mathcal{P}\setminus q}(s_{m'})$ and for some $q_A \in Q_A$, $D \cup q_A$ is a deadset of q. Intuitively, this means the B-states in the deadset that preserves the deadlock only change finitely often.

For $q \in Q$, we say that q *has alternation-bounded local deadlocks for* $c \in \mathbb{N}$ if the following holds for all $n \geq c$:

if $A \| B^n$ has a local deadlock in q
then $A \| B^n$ has an alternation-bounded local deadlock in q.

Theorem 5. *For conjunctive systems and process templates A, B, the cutoffs of Theorem 2 apply for non-fair runs, and the cutoff of Theorem 3 applies for strong-fair runs if every $q \in Q$ has alternation-bounded local deadlocks for the cutoff value. In particular, this implies that the parameterized local deadlock detection problem is decidable.*

Proof Sketch. Suppose in run x of $A \| B^n$, with n greater than the cutoff value, process p is locally deadlocked in local state $q \in Q$, and q has alternation-bounded local deadlocks. Then there exists an alternation-bounded run x' of $A \| B^n$ in which p is locally deadlocked in q. That is, either the local deadlock in x' eventually is preserved by a sequence of deadsets with unique restriction to B-states, or a number of processes that is bounded by the size of the largest deadset is sufficient to preserve the local deadlock in q. In the latter case, we are done. In the former case, based on the set D, the run x' can be simulated with the same constructions as in the proofs of Theorems 2 and 3. □

Sufficient Conditions for Alternation-bounded Local Deadlocks. In the following, we will identify four sufficient conditions that imply that a state q has alternation-bounded local deadlocks, and that can easily be checked directly on the process template.

Effectively 1-conjunctive states. We say that a state q is *effectively* 1-*conjunctive* if it is either 1-conjunctive or free.

Lemma 1. *If $q \in Q$ is effectively* 1-*conjunctive, then it has alternation-bounded local deadlocks for $c = 1$.*

Proof Sketch. If q is 1-conjunctive, then it has alternation-bounded local deadlocks since it has only a single deadset. If q is free, then a local deadlock in q is not possible, so the condition holds vacuously. □

In the reader-writer example from Section 1, all states except tw are effectively 1-conjunctive.

Relaxing 1-*conjunctiveness.* For $q \in Q$, let G_q be the set of non-trivial guards in transitions from q. We say that state q is *relaxed* 1-*conjunctive* if G_q only contains guards of the form $Q \setminus \{q_1, \ldots, q_k\}$, where either

- at most one of the q_i is from Q_B, or
- whenever more than one q_i is from Q_B, then G_q must also contain a guard of the form $Q \setminus \{q'_1, \ldots, q'_k, q_i\}$ for one of these q_i and where all q'_j are from Q_A.

Lemma 2. *If $q \in Q$ is relaxed* 1-*conjunctive, then it has alternation-bounded local deadlocks for $c = 1$.*

Proof Sketch. Note that the guards we allow on transitions from a relaxed 1-conjunctive state each have at most one state from Q_B that can block the transition. Thus q does not necessarily have a unique deadset, but for each deadset D the restriction to states of B is unique. Thus, every run that is locally deadlocked in q will be alternation-bounded. □

Alternation-free. For a given state $q \in Q$, let D be the set of all local states that disable one of the k-conjunctive guards, with $k > 1$, on transitions from q. Then we say that q *is alternation-free* if the following condition holds for at most one $q' \in D$:

there exists a cycle $C_{q'} = q', \dots, q' \in U$ with

- $q \notin C_{q'}$, and
- $\forall g \in G_{C_{q'}}$: $(q \in g \wedge \nexists g' \in G_q : g' \supseteq g)$.

Intuitively, this means that there is at most one state $q' \in D$ that is on a cycle that can be traversed while the local deadlock is preserved — and at least two such states would be needed to alternate between different deadsets.

In the reader-writer example from Section 1, state tw is alternation-free: i) $\{w, r\}$ is the set of states that disables the only guard that is not 1-conjunctive, and ii) all cycles that start and end in w contain also tw.

The following lemma directly follows from the explanation above.

Lemma 3. *If $q \in Q$ is alternation-free, then it has alternation-bounded local deadlocks for $c = 1$.*

Process templates with freely traversable lassos. While the three conditions above guarantee the existence of a fair alternation-bounded run if the original run was fair, the following condition in general returns a run that is not strong-fair. A *lasso lo* is a connected sequence of local states $q_0, \dots, q_i, \dots, q_k$ such that $q_0 = \mathsf{init}$ and $q_i, \dots, q_k$ is a cycle. We denote by G_{lo} the set of guards of the transitions on lo. We say that a lasso lo is *freely traversable* with respect to a state $q \in Q$ if it does not contain q and for every deadset D of q, every $g \in G_{lo}$ contains $D \cup \{q\}$. Intuitively, these conditions ensure that lo can be executed after the system has reached any of the (minimal) deadlock configurations for q.

Lemma 4. *If there exists a freely traversable lasso in B with respect to $q \in Q$, then q has alternation-bounded local deadlocks in non-fair runs for $c = \mathsf{maxD}_U + 2$.*

Proof Sketch. Suppose there exists a freely traversable lasso with respect to q, and x is a run where process p is locally deadlocked in q, where p is not enabled anymore after time m. Then we obtain an alternation-bounded locally deadlocked run x' by picking a deadset D of q with $D \subseteq \mathsf{set}_{\mathcal{P} \setminus p}(x_m)$ and for every $q' \in D$ a local run from x that is in q' at time m. Since $n \geq c = \mathsf{maxD}_U + 2$, there is at least one other process in $A \| B^n$. We replace the local run of this process with a local run that stays in init_B until m, and after m is the only process that

moves, along the freely traversable lasso we assumed to exist. Any further local runs stay in init_U forever.

Theorem 5 and Lemmas 1 to 4 allow us to analyze the process templates, state by state, and to conclude the existence of a small cutoff for local deadlock detection in certain cases. The lemmas provide sufficient but not necessary conditions for the existence of alternation-bounded cutoffs. They provide a template for obtaining small cutoffs in certain cases, and for a given application they may be refined depending on domain-specific knowledge.

3.5 Local Deadlock Detection under Infinite Alternation

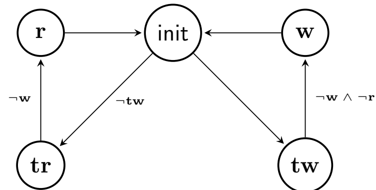
For systems that do not have alternation-bounded local deadlocks, it is very difficult to obtain cutoff results. For example, even in systems with a single 2-conjunctive and otherwise only 1-conjunctive guards, one can show that a cutoff based on the number of guards in general cannot exist. Moreover, the cutoff grows at least linearly in the number of states, or, more precisely, in the number of alternations between different deadsets that are necessary to traverse a cycle C_q for a state q from a deadset.

4 Verification of the Reader-Writer Example

We consider again the reader-writer example from Section 1, and show how our new results allow us to check correctness, find a bug, and check a fixed version.

With our results, we can for the first time check the given liveness property in a meaningful way, i.e., under the assumption of fair scheduling. Since all states in the process template have alternation-bounded local deadlocks for $c = 1$, by Theorems 3 and 5 the local deadlock detection cutoff for the system is $2|G_{B,B}| = 4$. No cutoff for this problem was known before. Moreover, compared to previous results we reduce the cutoff for global deadlock detection by recognizing that $k_1 = 3$ states can never be deadlocked, and $k_2 = 2$ additional states never appear in any guard. This reduces the cutoff to $2|B| - 2k_1 - 2k_2 = 10 - 6 - 4 = 0$, i.e., we detect that there are no global deadlocks without further analysis.

However, checking the system for local deadlocks shows that a local deadlock is possible: a process may forever be stuck in tw if the other processes move in a loop $(\mathsf{init}, tr, r)^\omega$ (and always at least one process is in r). To fix this, we can add an additional guard $\neg tw$ to the transition from init to tr, as shown in the process template to the right. For the resulting system, our results give a local deadlock detection cutoff of $2|G_{B,B}| = 6$, and a global deadlock detection cutoff of $2|B| - 2k_1 - 2k_2 - k_3 = 10 - 6 - 2 - 1 = 1$ (where k_3 is the number of states that do appear in guards and could be deadlocked themselves, but do not have a transition that is blocked by another process in the same state).



5 New Cutoff Results for Disjunctive Systems

In this section, we state our new cutoff results for disjunctive systems, and compare them to the previously known results in Table 2. Moreover, we show two extensions of the class of problems for which cutoffs are available:

1. systems where transitions are guarded with a conjunction of disjunctive guards (Section 5.4), and
2. two important classes of specifications that cannot be expressed in prenex indexed temporal logic (Section 5.5).

To state our results, we need the following additional definitions. Fix process templates A, B with $G = G_A \cup G_B$. Let $|B|_G = |\{q \in Q_B \mid \exists g \in G : q \in g\}|$. For a state $q \in Q_B$ in a disjunctive system, define $\mathsf{Enable}_q = \{q' \in Q \mid \exists (q, g, q'') \in \delta_B : q' \in g\}$, i.e., the set of states of A and B that enable a transition from q.

5.1 Linear-Time Properties

Theorem 6. *For disjunctive systems, process templates A, B, and k-indexed properties Φ_k:*

- $|G| + k + 1$ *and* $|B|_G + k + 1$ *are cutoffs in non-fair runs,*
- $|B|_G + |G| + k$ *and* $2|B|_G + k$ *are cutoffs in unconditionally fair runs.*

Proof Sketch. Given a run x of $A \| B^n$ where $x(B_1), \ldots, x(B_k)$ satisfy Φ_k, Emerson and Kahlon [15] showed how to construct a non-fair run y in the cutoff system that satisfies Φ_k. The run y includes the local runs $x(B_1), \ldots, x(B_k)$, and additional runs that ensure that all the transitions are enabled: for every state $q \in Q_B$ that appears in x and the local run that first visits q, we add the prefix of that local run up to q, and then let it stay in q forever. One additional local run may have to be copied from x to ensure that the resulting run is infinite. Thus, the resulting cutoff is $|B| + k + 1$ in non-fair runs.

Based on an analysis of the process template B, we can find better cutoffs: as a first option, we can statically check which states do appear in a guard, and conclude that only those may need to be copied. This reduces the cutoff to $|B|_G + k + 1$. Furthermore, since our goal is to enable all transitions, it is also sufficient to only copy a local run for one *representative* state of each guard (the one that is visited first in x). In this way, we need at most one additional process per guard in B, i.e., $|G| + k + 1$ also is a cutoff for non-fair runs.

Außerlechner et al. [6] gave a construction that builds on the steps explained above, and additionally preserves unconditional fairness in a given run. To this end, distinguish local states that appear finitely often or infinitely often in x. If state q appears infinitely often, there must be a cycle $C_q = q, \ldots, q$ in one of the local runs. To ensure fairness while always covering q, we add two copies of the shortest local path to q, and let the two processes take turns in moving through C_q (i.e., while one of them moves through C_q, the other one stays in q). If state q appears finitely often in x, we add a copy of the shortest path to

q, and identify the moment m_q when q appears for the last time in x. Instead of staying in q forever, we let the copied local run stay in q until m_q, and then move along the local path that leaves q at that time in x, until it reaches a state q' that appears infinitely often. From that point on, we let the process move in a fair way based on a cycle $C_{q'}$ taken from x. This original construction gives a cutoff of $2|B|+k-1$, since in the worst case all states appear infinitely often and we need two copies for each, but at least one of them must also appear infinitely often in the k processes that have to satisfy the specification.

Like in the non-fair case, an analysis of the template gives us better cutoffs. As a first approximation, we can again limit the construction to states in $|B|_G$, and obtain the cutoff $2|B|_G + k$ (now we can not assume that one of the states also appears in the k processes). Moreover, from the states in $|B|_G$ that appear infinitely often we can again chose one representative for each guard, and only add two local runs for each representative. This does not work for the processes that are visited finitely often, since we need to move them into an infinitely visited state to ensure fairness, and then need a different representative. To compute the cutoff, suppose f states from $|B|_G$ are visited finitely often, and i states infinitely often. From the latter, there are r states for which we added two local runs, with $r \leq |G|$ and $r \leq i$. Then we need at most $f + 2r + k$ local runs (including the k processes that satisfy the specification). However, we have $f \leq |B|_G - i$, and therefore $f + 2r + k \leq |B|_G - i + 2r + k \leq |B|_G + r + k \leq |B|_G + |G| + k$. □

5.2 Global Deadlock Detection

Let $\mathcal{N} = \{q \in Q_B \mid q \in \mathsf{Enable}_q\}$, and let $\mathcal{N}^*$ be the maximal subset (wrt. number of elements) of $\mathcal{N}$ such that $\forall q_i, q_j \in \mathcal{N}^* : q_i \notin \mathsf{Enable}_{q_j} \wedge q_j \notin \mathsf{Enable}_{q_i}$.

Theorem 7. *For disjunctive systems and process templates A, B, $|B|_G + |\mathcal{N}^*|$ is a cutoff for global deadlock detection.*

Proof Sketch. To construct a globally deadlocked run in the cutoff system, for each state from $\mathcal{N}$ that appears in the deadlock, we copy the according local run. To simulate the remaining part of x, we use the same construction as for fair runs in the proof of Thm. 6, except that local states that appear in the deadlock are considered to be visited infinitely often (and we don't need the fair extension of runs after reaching the state). Thus, the resulting run will be globally deadlocked, and all transitions up to the deadlock will be enabled. The number of local runs is bounded by $|\mathcal{N}| + f + i$, where i is the number of states from $|B|_G$ that appear in the deadlock and are not in $\mathcal{N}$, and f is the states from $|B|_G$ that appear in the run, but not in the deadlock. Since $f + i \leq |B|_G$ and $\mathcal{N}^*$ is the maximal subset of $\mathcal{N}$ that can appear together in a global deadlock, the number of needed local runs is bounded by $|\mathcal{N}^*| + |B|_G$. □

Remark. To compute $\mathcal{N}^*$ exactly, we need to find the smallest set of states in $\mathcal{N}$ that do not satisfy the additional condition. This amounts to finding the

minimum vertex cover (MVC) for the graph with vertices from $\mathcal{N}$ and edges from q_i to q_j if $q_i \in \mathsf{Enable}_{q_j}$.

5.3 Local Deadlock Detection

Theorem 8. *For disjunctive systems and process templates A, B:*

- $m + |G| + 1$ *is a cutoff for local deadlock detection in non-fair runs, where* $m = \max_{q \in Q_B^*}\{|\mathsf{Enable}_q|\}$ *for* $Q_B^* = \{q \in Q_B \mid |\mathsf{Enable}_q| < |B|\}$,
- $|B|_G + |G| + 1$ *and* $2|B|_G + 1$ *are cutoffs for local deadlock detection in unconditionally fair runs.*

Proof Sketch. Based on what we have already shown, the fair case is simpler: we copy the local runs of A and the deadlocked process, and for the other processes use the same construction as in the fair case of Thm. 6. The local deadlock is preserved since states that appear finitely often in the original run also appear finitely often in the constructed run, and the cutoffs are $2|B|_G + 1$ and $|B|_G + |G| + 1$.

For the non-fair case, we use a combination of the constructions for the fair and non-fair case from Thm. 6: if in run x a process is locally deadlocked in local state q, then for states in Enable_q that appear in x we use the construction for finitely appearing states in fair runs. For the remaining states, we use the non-fair construction, i.e., we find one representative per guard and stay there forever, except that representatives now can never be from Enable_q. The construction ensures that all transitions that are taken are enabled, and eventually all transitions from q are disabled. Since m gives a bound on the number of states that can be in Enable_q, the cutoff we get is $m + |G| + 1$. □

Table 2: Cutoff Results for Disjunctive Systems

		EK [15]	AJK [6]	our work
k-indexed LTL\X	non-fair	$\|B\| + k + 1$	$\|B\| + k + 1$	$\|G\| + k + 1$ and $\|B\|_G + k + 1$
k-indexed LTL\X	fair	-	$2\|B\| + k - 1$	$\|B\|_G + \|G\| + k$ and $2\|B\|_G + k$
Local Deadlock	non-fair	-	$\|B\| + 2$	$m + \|G\| + 1$, with $m < \|B\|$
Local Deadlock	fair	-	$2\|B\| - 1$	$\|B\|_G + \|G\| + 1$ and $2\|B\|_G + 1$
Global Deadlock		-	$2\|B\| - 1$	$\|B\| + \|\mathcal{N}^*\|$ with $\|\mathcal{N}^*\| < \|B\|$

5.4 Systems with Conjunctions of Disjunctive Guards

We consider systems where a transition can be guarded by a set of sets of states, interpreted as a conjunction of disjunctive guards. I.e., a guard $\{D_1, \ldots, D_n\}$ is

satisfied in a given global state if for all $i = 1, \ldots, n$, there exists another process in a state from D_i.

We observe that for this class of systems, most of the original proof ideas still work. For results that depend on the number of guards, we have to count the number of different conjuncts in guards.

Theorem 9. *For systems with conjunctions of disjunctive guards, cutoff results for disjunctive systems that do not depend on the number of guards still hold (first and second column of results in Table 2, and cutoffs in the third column that only refer to $|B|_G$ and constants).*

Cutoff results that depend on the number of guards (last column of Table 2) hold if we consider the number of conjuncts in guards instead. For results that additionally refer to some measure of the sets of enabling states (m and $|\mathcal{N}^*|$, respectively), we obtain a valid cutoff for systems with conjunctions of disjunctive guards if we replace this measure by $|B| - 1$.

In particular, the existence of a cutoff implies that the respective PMCP and parameterized deadlock detection problems are decidable.

Proof Ideas. The cutoff results that are independent of the number of guards still hold since all of the original proof constructions still work. To simulate a run x of a large system in a run y the cutoff system, one task is to make sure that all necessary transitions are enabled in the cutoff system. The construction that is used to do this works for conjunctions of disjunctive guards just as well. By a similar argument, deadlocks are preserved in the same way as for disjunctive systems.

For cutoffs that depend on the number of guards, transitions with conjunctions of disjunctive guards require us to use one representative for each conjunct in a guard, in the construction explained in the proof of Theorem 6.

Finally, the reductions of the cutoff based on the analysis of states that can or cannot appear together in a deadlock do not work in these extended systems, and we have to replace m and $|\mathcal{N}^*|$ by $|B| - 1$ in the cutoffs. The reason is that Enable_q is now not a set of states anymore, but a set of sets of states. A more detailed analysis based on this observation may be possible, but is still open.

5.5 Simultaneous Reachability of Target States

An important class of properties for parameterized systems asks for the reachability of a global state where all processes of type B are in a given local state q (compare Delzanno et al. [13]). This can be written in indexed $\mathsf{LTL}\backslash\mathsf{X}$ as $\mathsf{F}\,\forall i.q_i$, but is not expressible in the fragment where index quantifiers have to be in prenex form. We denote this class of specifications as TARGET. Similarly, repeated reachability of q by all states simultaneously can be written $\mathsf{GF}\,\forall i.q_i$, and is also not expressible in prenex form. We denote this class of specifications as REPEAT-TARGET.

Theorem 10 (Disjunctive Target and Repeat-Target). *For disjunctive systems: $|B|$ is a cutoff for checking* TARGET *and* REPEAT-TARGET.

In particular, the PMCP with respect to Target *and* Repeat-Target *in disjunctive systems is decidable.*

Proof Ideas. We can simulate a run x in a large system where all processes are in q at time m in the cutoff system by first moving one process into each state that appears in x before m, in the same order as in x. To make all processes reach q, we move them out of their respective states in the same order as they have moved out of them in x. For this construction, we need at most $|B|$ processes.

If in x the processes are repeatedly in q at the same time, then we can simulate this also in the cutoff system: if $m' > m$ is a point in time where this happens again, then we use the same construction as above, except that we consider all states that are visited between m and m', and we move to these states from q instead from init. The correctness argument is the same, however.

Finally, if the run with Repeat-Target should be fair, then we do not select *any* m' with the property above, but we choose it such that all processes move between m and m'. If the original run x is fair, then such an m' must exist. □

6 Conclusion

We have shown that better cutoffs for guarded protocols can be obtained by analyzing properties of the process templates, in particular the number and form of transition guards. We have further shown that cutoff results for disjunctive systems can be extended to a new class of systems with conjunctions of disjunctive guards, and to specifications Target and Repeat-Target, that have not been considered for guarded protocols before.

For conjunctive systems, previous works have treated local deadlock detection only for the restricted case of systems with 1-conjunctive guards. We have considered the general case, and have shown that it is very difficult — the cutoffs grow independently of the number of guards, and at least quadratically in the size of the process template. To circumvent this worst-case behavior, we have identified a number of conditions under which a small cutoff can be obtained even for systems that are not 1-conjunctive.

By providing cutoffs for several problems that were previously not known to be decidable, we have in particular proved their decidability.

Our work is inspired by applications in parameterized synthesis [8,22], where the goal is to automatically *construct* process templates such that a given specification is satisfied in systems with an arbitrary number of components. In this setting, deadlock detection and expressive specifications are particularly important, since *all* relevant properties of the system have to be specified.

Acknowledgements. We thank Ayrat Khalimov for fruitful discussions on guarded protocols, Martin Zimmermann for suggestions regarding a draft of this work, and the reviewers for insightful comments and suggestions. This work was supported by the German Research Foundation (DFG) through project Automatic Synthesis of Distributed and Parameterized Systems (JA 2357/2-1).

References

1. P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI*, volume 7737 of *LNCS*, pages 476–495. Springer, 2013. doi:10.1007/978-3-642-35873-9_28.
2. B. Aminof, S. Jacobs, A. Khalimov, and S. Rubin. Parameterized model checking of token-passing systems. In *VMCAI*, volume 8318 of *LNCS*, pages 262–281. Springer, 2014. doi:10.1007/978-3-642-54013-4_15.
3. B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR*, volume 8704 of *LNCS*, pages 109–124. Springer, 2014. doi:10.1007/978-3-662-44584-6_9.
4. Benjamin Aminof and Sasha Rubin. Model checking parameterised multi-token systems via the composition method. In *IJCAR*, volume 9706 of *LNCS*, pages 499–515. Springer, 2016. doi:10.1007/978-3-319-40229-1_34.
5. Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. Tight cutoffs for guarded protocols with fairness. *CoRR*, abs/1505.03273, 2015. Extended version with full proofs. URL: http://arxiv.org/abs/1505.03273.
6. Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. Tight cutoffs for guarded protocols with fairness. In *VMCAI*, volume 9583 of *LNCS*, pages 476–494. Springer, 2016. doi:10.1007/978-3-662-49122-5_23.
7. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.
8. Roderick Bloem, Swen Jacobs, and Ayrat Khalimov. Parameterized synthesis case study: AMBA AHB. In *SYNT*, volume 157 of *EPTCS*, pages 68–83, 2014. doi:10.4204/EPTCS.157.9.
9. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015. doi:10.2200/S00658ED1V01Y201508DCT013.
10. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000. doi:10.1007/10722167_31.
11. E. M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *CONCUR*, volume 3170 of *LNCS*, pages 276–291. Springer, 2004. doi:10.1007/978-3-540-28644-8_18.
12. E. M. Clarke, M. Talupur, and H. Veith. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*, volume 4963 of *LNCS*, pages 33–47. Springer, 2008. doi:10.1007/978-3-540-78800-3_4.
13. Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR*, volume 6269 of *LNCS*, pages 313–327. Springer, 2010. doi:10.1007/978-3-642-15375-4_22.
14. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982. doi:10.1016/0167-6423(83)90017-5.
15. E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, volume 1831 of *LNCS*, pages 236–254. Springer, 2000. doi:10.1007/10721959_19.
16. E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE Computer Society, 2003. doi:10.1109/LICS.2003.1210076.

17. E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *Foundations of Computer Science*, 14(4):527–549, 2003. doi:10.1142/S0129054103001881.
18. E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV*, volume 1102 of *LNCS*, pages 87–98. Springer, 1996. doi:10.1007/3-540-61474-5_60.
19. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782630.
20. Javier Esparza. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In *STACS*, volume 25 of *LIPIcs*, pages 1–10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.1.
21. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. doi:10.1145/146637.146681.
22. S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10:1–29, 2014. doi:10.2168/LMCS-10(1:12)2014.
23. A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 645–659. Springer, 2010. doi:10.1007/978-3-642-14295-6_55.
24. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Inf. and Comp.*, 117(1):1–11, 1995. doi:10.1006/inco.1995.1024.
25. I. Suzuki. Proving properties of a ring of finite state machines. *Inf. Process. Lett.*, 28(4):213–214, 1988. doi:10.1016/0020-0190(88)90211-6.

Refinement Types for Ruby

Milod Kazerounian[1], Niki Vazou[1], Austin Bourgerie[1], Jeffrey S. Foster[1], and Emina Torlak[2]

[1] University of Maryland, College Park, USA
{milod, nvazou, abourg, jfoster}@cs.umd.edu,
[2] University of Washington, Seattle, USA
emina@cs.washington.edu

Abstract. Refinement types are a popular way to specify and reason about key program properties. In this paper, we introduce RTR, a new system that adds refinement types to Ruby. RTR is built on top of RDL, a Ruby type checker that provides basic type information for the verification process. RTR works by encoding its verification problems into Rosette, a solver-aided host language. RTR handles mixins through assume-guarantee reasoning and uses just-in-time verification for metaprogramming. We formalize RTR by showing a translation from a core, Ruby-like language with refinement types into Rosette. We apply RTR to check a range of functional correctness properties on six Ruby programs. We find that RTR can successfully verify key methods in these programs, taking only a few minutes to perform verification.

Keywords: Ruby, Rosette, refinement types, dynamic languages

1 Introduction

Refinement types combine types with logical predicates to encode program invariants [32, 43]. For example, the following refinement type specification:

type : incr_sec , '(**Integer** x { 0 ≤ x < 60 }) → **Integer** r { 0 ≤ r < 60}'

describes a method incr_sec that increments a second. With this specification, incr_sec can only be called with integers that are valid seconds (between 0 and 59) and the method will always return valid seconds.

Refinement types were introduced to reason about simple invariants, like safe array indexing [43], but since then they have been successfully used to verify sophisticated properties including termination [39], program equivalence [9], and correctness of cryptographic protocols [28], in various languages (*e.g.,* ML [18], Racket [21], and TypeScript [40]).

In this paper, we explore refinement types for Ruby, a popular, object-oriented, dynamic scripting language. Our starting place is RDL [17, 30], a Ruby type system recently developed by one of the authors and his collaborators. We introduce RTR, a tool that adds refinement types to RDL and verifies them via a translation into Rosette [38], a solver-aided host language. Since Rosette is not object-oriented, RTR encodes Ruby objects as Rosette structs that store object

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 269–290, 2018.
https://doi.org/10.1007/978-3-319-73721-8_13

fields and an integer identifying the object's class. At method calls, RTR uses RDL's type information to statically overestimate the possible callees. When methods with refinement types are called, RTR can either translate the callee directly or treat it modularly by asserting the method preconditions and assuming the postcondition, using purity annotations to determine which fields (if any) the method may mutate. (§ 2)

In addition to standard object-oriented features, Ruby includes dynamic language features that increase flexibility and expressiveness. In practice, this introduces two key challenges in refinement type verification: *mixins*, which are Ruby code modules that extend other classes without direct inheritance, and *metaprogramming*, in which code is generated on-the-fly during runtime and used later during execution. The latter feature is particularly common in Ruby on Rails, a popular Ruby web development framework.

To meet these challenges, RTR uses two key ideas. First, RTR incorporates *assume-guarantee checking* [20] to reason about mixins. RTR verifies definitions of methods in mixins by assuming refinement type specifications for all undefined, external methods. Then, by dynamically intercepting the call that includes a mixin in a class, RTR verifies the appropriate class methods satisfy the assumed refinement types (§ 3.1). Second, RTR uses *just-in-time verification* to reason about metaprogramming, following RDL's just-in-time type checking [30]. In this approach, (refinement) types are maintained at run-time, and methods are checked against their types after metaprogramming code has executed but before the methods have been called (§ 3.2).

We formalized RTR by showing how to translate λ^{RB}, a core Ruby-like language with refinement types, into λ^{I}, a core verification-oriented language. We then discuss how to map the latter into Rosette, which simply requires encoding λ^{I}'s primitive object construct into Rosette structs and translating some control-flow constructs such as `return` (§ 4).

We evaluated RTR by using it to check a range of functional correctness properties on six Ruby and Rails applications. In total, we verified 31 methods, comprising 271 lines of Ruby, by encoding them as 1,061 lines of Rosette. We needed 73 type annotations. Verification took a total median time (over multiple trials) of 506 seconds (§ 5).

Thus, we believe RTR is a promising first step toward verification for Ruby.

2 Overview

We start with an overview of RTR, which extends the Ruby type checker RDL [30] with refinement types. In RTR, program invariants are specified with refinement types (§ 2.1) and checked by translation to Rosette (§ 2.2). We translate Ruby objects to Rosette structs (§ 2.3) and method calls to function calls (§ 2.4).

2.1 Refinement Type Specifications

Refinement types in RTR are Ruby types extended with logical predicates. For example, we can use RDL's **type** method to link a method with its specification:

```
type '(Integer x { 0 ≤ x < 60 }) → Integer r { 0 ≤ r < 60}'
def incr_sec (x) if (x==59) then 0 else x+1 end ; end
```

This type indicates the argument and result of incr_sec are integers in the range from 0 to 59. In general, refinements (in curly braces) may be arbitrary Ruby expressions that are treated as booleans, and they should be *pure*, *i.e.,* have no side effects, since effectful predicates make verification either complicated or imprecise [41]. As in RDL, the type annotation, which is a string, is parsed and stored in a global table which maintains the program's type environment.

2.2 Verification using Rosette

RTR checks method specifications by encoding their verification into Rosette [38], a solver-aided host language built on top of Racket. Among other features, Rosette can perform verification by using symbolic execution to generate logical constraints, which are discharged using Z3 [24].

For example, to check incr_sec , RTR creates the equivalent Rosette program:

```
(define ( incr_sec x) ( if (= x 59) 0 (+ x 1)))
(define-symbolic x_in integer?)
( verify #:assume (assert 0 ≤x < 60)
        #:guarantee (assert (let ([r ( incr_sec x)]) 0 ≤ r < 60)))
```

Here x_in is an integer *symbolic constant* representing an unknown, arbitrary integer argument. Rosette symbolic constants can range over the *solvable types* integers, booleans, bitvectors, reals, and uninterpreted functions. We use Rosette's **verify** function with assumptions and assertions to encode pre- and postconditions, respectively. When this program is run, Rosette searches for an x_in such that the assertion fails. If no such value exists, then the assertion is verified.

2.3 Encoding and Reasoning about Objects

We encode Ruby objects in Rosette using a *struct type*, i.e., a record. More specifically, we create a struct type **object** that contains an integer classid identifying the object's class, an integer objectid identifying the object itself, and a field for each instance variable of all objects encountered in the source Ruby program (similarly to prior work [19, 34]).

For example, consider a Ruby class **Time** with three instance variables @sec, @min, and @hour, and a method is_valid that checks all three variables are valid:

```
class Time
  attr_accessor :sec, :min, :hour

  def initialize (s, m, h) @sec = s; @min = m; @hour = h; end

  type '() → bool'
  def is_valid 0 ≤ @sec < 60 ∧ 0 ≤ @min < 60 ∧0 ≤ @hour < 24; end
end
```

RTR observes three fields in this program, and thus it defines:

```
(struct object ([ classid ][ objectid ]
                [@sec #:mutable] [@min #:mutable] [@hour #:mutable]))
```

Here **object** includes fields for the class ID, object ID, and the three instance variables. Note since **object**'s fields are statically defined, our encoding cannot handle dynamically generated instance variables, which we leave as future work.

RTR then translates Ruby field reads or writes as getting or setting, respectively, **object**'s fields in Rosette. For example, suppose we add a method mix to the **Time** class and specify it is only called with and returns valid times:

```
type :mix, '(Time t1 { t1. is_valid }, Time t2 { t2. is_valid },
             Time t3 { t3. is_valid }) → Time r { r. is_valid }'
def mix(t1,t2,t3) @sec = t1.sec; @min = t2.min; @hour = t3.hour; self; end
```

Initially, type checking fails because the getters' and setters' (*e.g.,* sec and sec=) types are unknown. Thus, we add those types:

```
type :sec,  '()          → Integer i   { i == @sec }'
type :sec=, '(Integer i) → Integer out { i == @sec }'
```

(Note these annotations can be generated automatically using our approach to metaprogramming, described in § 3.2.) This allows RTR to proceed to the translation stage, which generates the following Rosette function:

```
(define (mix self t1 t2 t3)
  (set−object−@sec! self (sec t1))
  (set−object−@min! self (min t2))
  (set−object−@hour! self (hour t3))
  self )
```

(Asserts, assumes, and verify call omitted.) Here (**set**−**object**−x! y w) writes w to the x field of y and the field selectors sec, min, and hour are uninterpreted functions. Note that **self** turns into an explicit additional argument in the Rosette definition. Rosette then verifies this program, thus verifying the original Ruby mix method.

2.4 Method Calls

To translate a Ruby method call e.m(e1, .., en), RTR needs to know the callee, which depends on the runtime type of the receiver e. RTR uses RDL's type information to overapproximate the set of possible receivers. For example, if e has type A in RDL, then RTR translates the above as a call to A.m. If e has a union type, RTR emits Rosette code that branches on the potential types of the receiver using **object** class IDs and dispatches to the appropriate method in each branch. This is similar to class hierarchy analysis [16], which also uses types to determine the set of possible method receivers and construct a call graph.

Once the method being called is determined, we translate the call into Rosette. As an example, consider a method to_sec that converts **Time** to seconds, after it calls the method incr_sec from § 2.1.

```
type '(Time t { t. is_valid }) → Integer r { 0≤r<90060 }'
def to_sec(t) incr_sec(t.sec) + 60 * t.min + 3600 * t.hour; end
```

RTR's translation of to_sec could simply call directly into incr_sec's translation. This is equivalent to inlining incr_sec's code. However, inlining is not always possible or desirable. A method's code may not be available because the method comes from a library, is external to the environment (§ 3.1), or has not been defined yet (§ 3.2). The method might also contain constructs that are difficult to verify, like diverging loops.

Instead, RTR can model the method call using the programmer provided method specification. To precisely reason with only a method's specification, RTR follows Dafny [22] and treats pure and impure methods differently.

Pure methods. Pure methods have no side effects and return the same result for the same inputs, satisfying the congruence property $\forall x, y. x = y \Rightarrow m(x) = m(y)$ for a given method m. Thus, pure methods can be encoded using Rosette's uninterpreted functions. The method incr_sec is indeed pure, so we can label it as such:

```
type :incr_sec, '(Integer x { 0≤x<60 }) → Integer r { 0≤r<60 }', :pure
```

With the **pure** label, the translation of to_sec treats incr_sec as an uninterpreted function. Furthermore, it asserts the precondition 0≤x<60 and assumes the postcondition 0≤r<60, which is enough to verify to_sec.

Impure methods. Most Ruby methods have side effects and thus are not pure. For example, consider incr_min, a mutator method that adds a minute to a **Time**:

```
type '(Time t { t. is_valid ∧ t.min < 59 }) → Time r { r. is_valid }',
  modifies: { t: @min, t: @sec }
def incr_min(t)
  if t.sec<59 then t.sec=incr_sec(t.sec) else t.min+=1; t.sec=0 end
  return t
end
```

A translated call to incr_min generates a fresh symbolic value as the method's output and assumes the method's postcondition on that value. Because the method may have side effects, the **modifies** label is used to list all fields of inputs which may be modified by the method. Here, a translated call to incr_min will *havoc* (set to fresh symbolic values) t's @min and @sec fields.

We leave support for other kinds of modifications (e.g., global variables, transitively reachable fields), as well as enforcing the **pure** and **modifies** labels, as future work.

3 Just-In-Time Verification

Next, we show how RTR handles code with dynamic bindings via mixins (§ 3.1) and metaprogramming (§ 3.2).

3.1 Mixins

Ruby implements mixins via its *module* system. A Ruby module is a collection of method definitions that are added to any class that **include**s the module at runtime. Interestingly, modules may refer to methods that are not defined in the module but will ultimately be defined in the including class. Such incomplete environments pose a challenge for verification.

Consider the following method that has been extracted and simplified from the `Money` library described in § 5.

```
module Arithmetic
  type '(Integer x)→ Float r { r==x/value }'
  def div_by_val (x) x/value; end
end
```

The module method div_by_val divides its input x by value. RTR's specification for / requires that value cannot be 0.

Notice that value is not defined in **Arithmetic**. Rather, it must be defined wherever **Arithmetic** is included. Therefore, to proceed with verification in RTR, the programmer must provide an annotation for value:

```
type :value, '() → Float v { 0 < v }', :pure
```

Using this annotation, RTR can verify div_by_value. Then when **Arithmetic** is included in another class, RTR verifies value's refinement type. For example, consider the following code:

```
class Money
  include Arithmetic
  def value()
    if (@val > 0) then (return @val) else (return 0.01) end
  end
end
```

RTR dynamically intercepts the call to **include** and then applies the type annotations for methods not defined in the included module, in this case verifying value against the annotation in **Arithmetic**. Thus, RTR follows an assume-guarantee paradigm [20]: it assumes value's annotation while verifying div_by_val and then guarantees the annotation once value is defined.

3.2 Metaprogramming

Metaprogramming in Ruby allows new methods to be created and existing methods to be redefined on the fly, posing a challenge for verification. RTR addresses this challenge using just-in-time checking [30], in which, in addition to code, method annotations can also be generated dynamically.

We illustrate the just-in-time approach using an example from `Boxroom`, a Rails app for managing and sharing files in a web browser (§ 5). The app defines a class **UserFile** that is a Rails *model* corresponding to a database table:

```
class UserFile < ActiveRecord::Base
  belongs_to :folder
  ... type '(Folder target) → Bool b { folder == target }'
    def move(target) folder = target; save!; end...
end
```

Here calling belongs_to tells Rails that every **UserFile** is associated with a folder (another model). The move method updates the associated folder of a **UserFile** and saves the result to the database. We annotate move to specify that the **UserFile**'s new folder should be the same as move's argument.

This method and its annotation are seemingly simple, but there is a problem. To verify move, RTR needs an annotation for the folder= method, which is not statically defined. Rather, it is dynamically generated by the call to belongs_to.

To solve this problem in RTR, we instrument belongs_to to generate type annotations for the setter (and getter) method, as follows:

```
module ActiveRecord::Associations::ClassMethods
  pre(:belongs_to) do |*args|
    name = args[0].to_s
    cname = name.camelize
    type '#{name}', '() → #{cname} c', :pure
    type '#{name}=', '(#{cname} i) →#{cname} o { #{name} == i }'
    true
  end
end
```

We call **pre**, an RDL method, to define a precondition *code block* (*i.e.,* an anonymous function) which will be executed on each call to belongs_to. First, the block sets name and cname to be the string version of the first argument passed to belongs_to and its camelized representation, respectively. Then, we create types for the name and name= methods. Finally, we return true so the contract will succeed. In our example, this code generates the following two type annotations upon the call to belongs_to:

```
type 'folder', '() → Folder c', :pure
type 'folder=', '(Folder i) → Folder o { folder == i }'
```

With these annotations, verification of the initial move method succeeds.

4 From Ruby to Rosette

In this section, we formally describe our verifier and the translation from Ruby to Rosette. We start (§ 4.1) by defining λ^{RB}, a Ruby subset that is extended with refinement type specifications. We give (§ 4.2) a translation from λ^{RB} to an intermediate language λ^{I}, and then (§ 4.3) we discuss how λ^{I} maps to a Rosette program. Finally (§ 4.5), we use this translation to construct a verifier for Ruby.

Constants	$c ::= \texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid 0, 1, -1, \ldots$
Expressions	$e ::= c \mid x \mid x{:=}e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ ;\ e$
	$\mid \texttt{self} \mid f \mid f{:=}e \mid e.m(\overline{e}) \mid A.\texttt{new} \mid \texttt{return}(e)$
Refined Types	$t ::= \{x : A \mid e\ \}$
Program	$P ::= \cdot \mid d, P \mid a, P$
Definition	$d ::= \texttt{def}\ A.m(\overline{t}){::}t; l = e$
Annotation	$a ::= A.m :: (\overline{t}) \rightarrow t\ ;\ l$ with $l \neq \texttt{exact}$
Labels	$l ::= \texttt{exact} \mid \texttt{pure} \mid \texttt{modifies}[\overline{x.f}]$

$x \in$ var ids, $f \in$ field ids, $m \in$ meth ids, $A \in$ class ids

Fig. 1. Syntax of the Ruby Subset λ^{RB}.

Values	$w ::= c \mid \texttt{object}(i,\ i,\ \overline{[f\ w]})$
Expressions	$u ::= w \mid x \mid x{:=}u \mid \texttt{if}\ u\ \texttt{then}\ u\ \texttt{else}\ u \mid u\ ;\ u$
	$\mid \texttt{let}\ (\overline{[x\ u]})\ \texttt{in}\ u \mid x(\overline{u}) \mid \texttt{assert}(u)$
	$\mid \texttt{assume}(u) \mid \texttt{return}(u) \mid \texttt{havoc}(x.f) \mid x.f := u \mid x.f$
Program	$Q ::= \cdot \mid d, Q \mid v, Q$
Definition	$d ::= \texttt{define}\ x(\overline{x}) = u \mid \texttt{define-sym}(x,\ A)$
Verification Query	$v ::= \texttt{verify}(\overline{u} \Rightarrow u)$

$x \in$ var ids, $f \in$ field ids, $A \in$ types, $i \in$ integers

Fig. 2. Syntax of the Intermediate Language λ^{I}.

4.1 Core Ruby λ^{RB} and Intermediate Representation λ^{I}

λ^{RB}. Figure 1 defines λ^{RB}, a core Ruby-like language with refinement types. *Constants* consist of `nil`, booleans, and integers. *Expressions* include constants, variables, assignment, conditionals, sequences, and the reserved variable `self`, which refers to a method's receiver. Also included are references to an instance variable f and instance variable assignment; we note that in actual Ruby, field names are preceded by a "@". Finally, expressions include method calls, constructor calls $A.$`new` which create a new instance of class A, and return statements.

Refined types $\{x : A \mid e\ \}$ refine the basic type A with the predicate e. The basic type A is used to represent both user defined and built-in classes including nil, booleans, integers, floats, etc. The refinement e is a *pure, boolean valued* expression that may refer to the refinement variable x. In the interest of greater simplicity for the translation, we require that `self` *does not* appear in refinements e; however, extending the translation to handle this is natural, and our implementation allows for it. Sometimes we simplify the trivially refined type $\{x : A \mid \texttt{true}\ \}$ to just A.

A *program* is a sequence of method definitions and type annotations over methods. A method definition `def` $A.m(\{x_1 : A_1 \mid e_1\ \}, \ldots, \{x_n : A_n \mid e_n\ \}){::}t; l =$

e defines the method $A.m$ with arguments $x_1, \ldots, x_n$ and body e. The type specification of the definition is a dependent function type: each argument binder x_i can appear inside the arguments' refinements types e_j for all $1 \leq j \leq n$, and can also appear in the refinement of the result type t. A method type annotation $A.m :: (\bar{t}) \to t \; ; \; l$ binds the method named $A.m$ with the dependent function type $(\bar{t}) \to t$. λ^{RB} includes both method annotations and method definitions because annotations are used when a method's code is not available, *e.g.,* in the cases of library methods, mixins, or metaprogramming.

A *label* l can appear in both method definitions and annotations to direct the method's translation into Rosette as described in § 2.4. The label `exact` states that a called method will be exactly translated by using the translation of the body of the method. Since method type annotations do not have a body, they cannot be assigned the `exact` label. The `pure` label indicates that a method is pure and thus can be translated using an uninterpreted function. Finally, the `modifies`$[\overline{x.f}]$ label is used when a method is impure, *i.e.,* it may modify its inputs. As we saw earlier, the list $\overline{x.f}$ captures all the argument fields which the method may modify.

λ^I. Figure 2 defines λ^I, a core verification-oriented language that easily translates to Rosette. λ^{RB} methods map to λ^I functions, and λ^{RB} objects map to a special `object` struct type. λ^I provides primitives for creating, altering, and referencing instances of this type. *Values* in λ^I consist of *constants* c (defined identically to λ^{RB}) and `object`$(i_1,\ i_2,\ [f_1\ w_1] \ldots [f_n\ w_n])$, an instantiation of an `object` type with class ID i_1, object ID i_2, and where each field f_i of the `object` is bound to value w_i. *Expressions* include `let` bindings (`let` $(\overline{[x_i\ u_i]})$ `in` u) where each x_i may appear free in u_j if $i < j$. They also include function calls, `assert`, `assume`, and `return` statements, as well as `havoc`$(x.f)$, which mutates x's field f to a fresh symbolic value. Finally, they include field assignment $x.f := u$ and field reads $x.f$.

A *program* is a series of definitions and verification queries. A *definition* is a function definition or a symbolic definition `define-sym`$(x,\ A)$, which binds x to either a fresh symbolic value if A is a solvable type (e.g., boolean, integer; see § 2.2) or a new `object` with symbolic fields defined depending on the type of A. Finally, a verification query `verify`$(\overline{u} \Rightarrow u)$ checks the validity of u assuming $\overline{u}$.

4.2 From λ^{RB} to λ^I

Figure 3 defines the translation function $e \rightsquigarrow u$ that maps expressions (and programs) from λ^{RB} to λ^I.

Global States. The translation uses sets $\mathcal{M}, \mathcal{U}$, and $\mathcal{F}$, to ensure all the methods, uninterpreted functions, and fields are well-defined in the generated λ^I term:

$$\mathcal{M} ::= A_1.m_1, \ldots, A_n.m_n \qquad \mathcal{U} ::= A_1.m_1, \ldots, A_n.m_n \qquad \mathcal{F} ::= f_1, \ldots, f_n$$

Expression Translation $\boxed{e \rightsquigarrow u}$

$$\frac{}{c \rightsquigarrow c}\ \text{T-Const} \qquad \frac{}{x \rightsquigarrow x}\ \text{T-Var} \qquad \frac{e_1 \rightsquigarrow u_1 \quad e_2 \rightsquigarrow u_2}{e_1\ ;\ e_2 \rightsquigarrow u_1\ ;\ u_2}\ \text{T-Seq}$$

$$\frac{e_1 \rightsquigarrow u_1 \quad e_2 \rightsquigarrow u_2 \quad e_3 \rightsquigarrow u_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \texttt{if}\ u_1\ \texttt{then}\ u_2\ \texttt{else}\ u_3}\ \text{T-If} \qquad \frac{}{\texttt{self} \rightsquigarrow \mathit{self}}\ \text{T-Self}$$

$$\frac{e \rightsquigarrow u}{x{:=}e \rightsquigarrow x{:=}u}\ \text{T-VarAssn} \qquad \frac{e \rightsquigarrow u}{\texttt{return}(e) \rightsquigarrow \texttt{return}(u)}\ \text{T-Ret}$$

$$\frac{f \in \mathcal{F}}{f \rightsquigarrow \mathit{self}.f}\ \text{T-Inst} \qquad \frac{f \in \mathcal{F} \quad e \rightsquigarrow u}{f{:=}e \rightsquigarrow \mathit{self}.f := u}\ \text{T-InstAssn}$$

$$\frac{\texttt{classId}(A) = i_c \quad \texttt{freshID}(i_o) \quad f_i \in \mathcal{F}}{A.\texttt{new} \rightsquigarrow \texttt{object}(i_c,\ i_o,\ [f_1\ \texttt{nil}]\ldots[f_{|\mathcal{F}|}\ \texttt{nil}])}\ \text{T-New}$$

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \quad \texttt{exact} = \texttt{labelOf}(A.m)\\ A_m \in \mathcal{M} \quad e_F \rightsquigarrow u_F \quad e_i \rightsquigarrow u_i\end{array}}{e_F.m(\overline{e}) \rightsquigarrow A_m(u_F, \overline{u})}\ \text{T-Exact}$$

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \qquad \texttt{pure} = \texttt{labelOf}(A.m)\\ A_m \in \mathcal{U} \qquad \texttt{freshVar}(x, r)\\ \texttt{specOf}(A.m) = (\{x : A_x \mid e_x\ \}) \to \{r : A_r \mid e_r\ \}\\ e_F \rightsquigarrow u_F \quad e \rightsquigarrow u \quad e_x \rightsquigarrow u_x \quad e_r \rightsquigarrow u_r\end{array}}{e_F.m(e) \rightsquigarrow \texttt{let}\ ([x\ u][r\ A_m(u_F, a)])\ \texttt{in}\ \texttt{assert}(u_x)\ ;\ \texttt{assume}(u_r)\ ;\ r}\ \text{T-Pure1}$$

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \qquad \texttt{modifies}[p] = \texttt{labelOf}(A.m)\\ \texttt{specOf}(A.m) = (\{x : A_x \mid e_x\ \}) \to \{r : A_r \mid e_r\ \}\\ h_x = \{u.f \mid f \in \mathcal{F},\ x.f \in p\} \quad h_F = \{u_F.f \mid f \in \mathcal{F},\ \texttt{self}.f \in p\}\\ \texttt{freshVar}(x, r) \quad e_F \rightsquigarrow u_F \quad e \rightsquigarrow u \quad e_x \rightsquigarrow u_x \quad e_r \rightsquigarrow u_r\end{array}}{e_F.m(e) \rightsquigarrow \begin{array}{l}\texttt{let}\ ([x\ u])\ \texttt{in}\ \texttt{define-sym}(r,\ A_r);\\ \texttt{assert}(u_x)\ ;\ \texttt{havoc}(h_F \cup h_x)\ ;\ \texttt{assume}(u_r)\ ;\ r\end{array}}\ \text{T-Impure1}$$

Program Translation $\boxed{P \rightsquigarrow Q}$

$$\frac{}{\cdot \rightsquigarrow \cdot}\ \text{T-Emp} \qquad \frac{P \rightsquigarrow Q}{A.m :: (x_1{:}t_1, \ldots, x_n{:}t_n) \to t\ ;\ l, P \rightsquigarrow Q}\ \text{T-Ann}$$

$$\frac{\begin{array}{c}t_i = \{x_i : A_{x_i} \mid e_{x_i}\ \} \qquad t = \{r : A_r \mid e_r\ \}\\ e \rightsquigarrow u \quad e_{x_i} \rightsquigarrow u_{x_i} \quad e_r \rightsquigarrow u_r \quad P \rightsquigarrow Q \quad 1 \leq i \leq n\end{array}}{\texttt{def}\ A.m(t_1, \ldots, t_n){::}t; l = e, P \rightsquigarrow \begin{array}{l}\texttt{define}\ A_m(\mathit{self}, x_1, \ldots, x_n) = u;\\ \texttt{define-sym}(self,\ A);\\ \texttt{define-sym}(x_i,\ A_{x_i});\\ \texttt{verify}(u_{x_1}, \ldots, u_{x_n} \Rightarrow u_r)\ ;\ Q\end{array}}\ \text{T-Def}$$

Fig. 3. Translation from λ^{RB} to λ^{I}. For simplicity rules T-Pure1 and T-Impure1 assume single argument methods.

In the translation rules, we use the standard set operations $x \in \mathcal{X}$ and $|\mathcal{X}|$ to check membership and size of the set $\mathcal{X}$. Thus, the translation relation is defined over these sets: $\mathcal{M},\mathcal{U},\mathcal{F} \vdash e \rightsquigarrow u$. Since the rules do not modify these environments, in Figure 3 we simplify the rules to $e \rightsquigarrow u$. Note that even though the rules "guess" these environments by making assumptions about which elements are members of the sets, in an algorithmic definition the rules can be used to construct the sets.

Expressions. The rules T-Const and T-Var are identity while the rules T-If, T-Seq, T-Ret, and T-VarAssn are trivially inductively defined. The rule T-Self translates `self` into the special *variable* named *self* in λ^I. The *self* variable is always in scope, since each λ^{RB} method translates to a λ^I function with an explicit first argument named *self*. The rules T-Inst and T-InstAssn translate a reference from and an assignment to the instance variable f, to a read from and write to, respectively, the field f of the variable *self*. Moreover, both the rules assume the field f to be in global field state $\mathcal{F}$. The rule T-New translates from a constructor call A.`new` to an `object` instance. The `classId`(A) function in the premise of this rule returns the class ID of A. The `freshID`(i_o) predicate ensures the new `object` instance has a fresh object ID. Each field of the new `object`, $f_1, \ldots, f_{|\mathcal{F}|}$, is initially bound to `nil`.

Method Calls. To translate the λ^{RB} method call $e_F.m(\overline{e})$, we first use the function `typeOf`(e_F) to type e_F via RDL type checking [30]. If e_F is of type A, we split cases of the method call translation based on the value of `labelOf`$(A.m)$, the label specified in the annotation of $A.m$ (as informally described in § 2.4).

The rule T-Exact is used when the label is `exact`. The receiver e_F is translated to u_F which becomes the first (*i.e.,* the *self*) argument of the function call to A_m. Moreover, $A.m$ is assumed to be in the global method name set $\mathcal{M}$ since it belongs to the transitive closure of the translation.

We note that for the sake of clarity, in the T-Pure1 and T-Impure1 rules, we assume that the method $A.m$ takes just one argument; the rules can be extended in a natural way to account for more arguments. The rule T-Pure1 is used when the label is `pure`. In this case, the call is translated as an invocation to the uninterpreted function A_m, so $A.m$ should be in the global set of uninterpreted functions $\mathcal{U}$. The specification `specOf`$(A.m)$ of the method is also enforced. Let $(\{x : A_x \mid e_x\}) \rightarrow \{r : A_r \mid e_r\}$ be the specification. We assume that the binders in the specification are α-renamed so that the binders x and r are fresh. We use x and r to bind the argument and the result, respectively, to ensure, via A-normal form conversion [33], that they will be evaluated exactly once, even though x and r may appear many times in the refinements. To enforce the specification, we assert the method's precondition e_x and assume the postcondition e_r.

If a method is labeled with `modifies`$[p]$ then the rule T-Impure1 is applied. We locally define a new symbolic object as the return value, and we `havoc` the fields of all arguments (including *self*) specified in the `modifies` label, thereby assigning these fields to new symbolic values. Since we do not translate the called method at all, no global state assumptions are made.

Programs. Finally, we use the translation relation to translate programs from λ^{RB} to λ^{I}, *i.e.,* $P \rightsquigarrow Q$. The rule T-Ann discards type annotations. The rule T-Def translates a method definition for $A.m$ to the function definition A_m that takes the additional first argument *self*. The rule also considers the declared type of $A.m$ and instantiates a symbolic value for every input argument. Finally, all refinements from the inputs and output of the method type are translated and the derived verification query is made.

4.3 From λ^{I} to Rosette

We write $Q \twoheadrightarrow R$ to encode the translation of the λ^{I} program Q to the Rosette program R. This translation is straightforward, since λ^{I} consists of Rosette extended with some macros to encode Ruby-verification specific operators, like `define-sym` and `return`. In fact, in the implementation of the translation (§ 5), we used Racket's macro expansion system to achieve this final transformation.

Handling objects. λ^{I} contains multiple constructs for defining and altering objects, which are expanded in Rosette to perform the associated operations over `object` structs. The expressions `object`$(i_c,\ i_o,\ \overline{[f\ w]})$ and `havoc`$(x.f)$, and the definition `define-sym`$(x,\ A)$, all described in § 4.1, are expanded to perform the corresponding operations over values of the `object` struct type.

Control Flow. Macro expansion is used to translate `return` and `assume` statements, and exceptions into Rosette, since those forms are not built-in to the language. To encode `return`, we expand every function definition in λ^{I} to keep track of a local variable `ret`, which is initialized to a special `undefined` value and returned at the end of the function. We transform every statement `return`(e) to update the value of `ret` to e. Then, we expand every expression u in a function to `unless-done`(u), which checks the value of `ret`, proceeding with u if `ret` is `undefined` or skipping u if there is a return value.

We used the encoding of `return` to encode more operators. For example, `assume` is encoded in Rosette as a macro that returns a special `fail` value when assumptions do not hold. The verification query then needs to be updated with the condition that `fail` is not returned. A similar expansion is used to encode and propagate exceptions.

4.4 Primitive Types

λ^{RB} provides constructs for functions, assignments, control flow, etc, but does not provide the theories required to encode interesting verification properties that, for example, reason about booleans and numbers. On the other hand, Rosette is a verification oriented language with special support for common theories over built-in datatypes, including booleans, numeric types, and vectors. To bridge this gap, we encode certain Ruby expressions, such as constants c in λ^{RB}, into Rosette's corresponding built-in datatypes.

Equality and Booleans. To precisely reason about equality, we encode Ruby's `==` method over arbitrary objects using the object class' == method if one is defined. If the class inherits this method from Ruby's top class, *BasicObject*, then we encode == using Rosette's equality operator `equal?` to check equality of object IDs. We encode Ruby's booleans and operations over them as Rosette's respective booleans and their operators.

Integers and Floats. By default, we encode Ruby's infinite-precision `Integer` and `Float` objects as Rosette's built-in infinite-precision `integer` and `real` datatypes, respectively. The infinite-precision encoding is efficient and precise, but it may result in undecidable queries involving non-linear arithmetic or loops. To perform (bounded) verification in such cases, we provide, via a configuration flag, the option of encoding Ruby's integers as Rosette's built-in finite sized bitvectors.

Arrays. Finally, we provide a special encoding for Ruby's arrays, which are commonly used both for storing arbitrarily large random-access data and to represent mixed-type tuples, stacks, queues, etc. We encode Ruby's arrays as a Rosette struct composed of a fixed-size vector and an integer representing the current size of the Ruby array. Because we used fixed-size vectors, we can only perform bounded verification over arrays. On the other hand, we avoid the need for loop invariants for iterators and reasoning over array operations can be more efficient.

4.5 Verification of λ^{RB}

We define a verification algorithm RTR^λ that, given a λ^{RB} program P, checks if all the definitions satisfy their specifications. The pseudo-code for this algorithm is shown below:

def $\mathsf{RTR}^\lambda(P)$
  $(\mathcal{F}, \mathcal{U}, \mathcal{M}) := \mathsf{guess}(P)$
  for $(f \in \mathcal{F})$: add field f to **object struct**
  for $(u \in \mathcal{U})$: define uninterpreted function u
  $P \rightsquigarrow Q \twoheadrightarrow R$
  return if (*valid*(R)) **then** SAFE **else** UNSAFE
end

First, we `guess` the proper translation environments. In practice (as discussed in § 4.2), we use the translation of P to generate the minimum environments for which translation of P succeeds. We define an `object` struct in Rosette containing one field for each member of $\mathcal{F}$, and we define an uninterpreted function for each method in $\mathcal{U}$. Next, we translate P to a λ^I program Q via $P \rightsquigarrow Q$ (§ 4.2) and Q to a the Rosette program R, via $Q \twoheadrightarrow R$ (§ 4.3). Finally, we run the Rosette program R. The initial program P is *safe*, *i.e.,* no refinement type specifications are violated, if and only if the Rosette program R is *valid*, *i.e.,* all the `verify` queries are valid.

We conclude this section with a discussion of the RTR^λ verifier.

RTR^{λ} is Partial. There exist expressions of λ^{RB} that fail to translate into a λ^{I} expression. The translation requires at each method call $e_F.m(\overline{e})$ that the receiver has a class type A. There are two cases where this requirement fails: (1) e_F has a union type or (2) type checking fails and so e_F has no type. In our implementation (§ 5), we extend the translation to handle the first two cases. Handling for (1) is outlined in § 2.4. Case (2) can be caused by either a type error in the program or a lack of typing information for the type checker. Translation cannot proceed in either case.

RTR^{λ} may Diverge. The translation to Rosette always terminates. All translation rules are inductively defined: they only recurse on syntactically smaller expressions or programs. Also, since the input program is finite, the minimum global environments required for translation are also finite. Finally, all the helper functions (including the type checking `typeOf`$(\cdot)$) do terminate.

Yet, verification may diverge, as the execution of the Rosette program may diverge. Specifications can encode arbitrary expressions, thus it is possible to encode undecidable verification queries. Consider, for instance, the following contrived Rosette program in which we attempt to verify an assertion over a recursive method:

```
(define (rec x) (rec x))
(define-symbolic b boolean?)
(verify (rec b))
```

Rosette attempts to symbolically evaluate this program, and thus diverges.

RTR^{λ} is Incomplete. Verification is incomplete and its precision relies on the precision of the specifications. For instance, if a pure method $A.m$ is marked as impure, the verifier will not prove the congruence axiom.

RTR^{λ} is Sound. If the verifier decides that the input program is safe, then all definitions satisfy their specifications, assuming that (1) all the refinements are pure boolean expressions and (2) all the labels are sound (i.e., methods match the specifications implied by the labels). The assumption (1) is required since verification under diverging (let alone effectful) specifications is difficult [41]. The assumption (2) is required since our translation encodes pure methods as uninterpreted functions, while for the impure methods it havocs only the unprotected arguments.

5 Evaluation

We implemented the Ruby refinement type checker RTR[3] by extending RDL [30] with refinement types. Table 1 summarizes the evaluation of RTR.

[3] Code available at: `https://github.com/mckaz/vmcai-rdl`

Benchmarks. We evaluate RTR on six popular Ruby libraries:

– `Money` [6] performs currency conversions over monetary quantities and relies on mixin methods,
– `BusinessTime` [3] performs time calculations in business hours and days,
– `Unitwise` [7] performs various unit conversions,
– `Geokit` [4] performs calculations over locations on Earth,
– `Boxroom` [2] is a Rails app for sharing files in a web browser and uses metaprogramming, and
– `Matrix` [5] is a Ruby standard library for matrix operations.

For verification, we forked the original Ruby libraries and provided manually written method specifications in the form of refinement types. The forked repositories are publicly available [8]. Experiments were conducted on a machine with a 3 GHz Intel Core i7 processor and 16 GB of memory.

We chose these libraries because they combine Ruby-specific features challenging for verification, like metaprogramming and mixins, with arithmetic-heavy operations. In all libraries we verify both (1) *functional correctness of arithmetic operations* (*e.g.,* no division-by-zero, the absolute value of a number should not be negative) and (2) *data-specific arithmetic invariants* (*e.g.,* integers representing months should always be in the range from `1` to `12` and a `data` value added to an aggregate should always fall between maintained `@min` and `@max` fields). In the `Matrix` library, we verify a matrix multiplication method, checking that multiplying a matrix with r rows by a matrix with c columns yields a matrix of size $r \times c$. Note this method makes extensive use of array operations, since matrices are implemented as an array of arrays.

Quantitative Evaluation. Table 1 summarizes our evaluation quantitatively. For each application, we list every verified **Method**. In our experiment, we focused on methods with interesting arithmetic properties.

The **Ruby LoC** column gives the size of the verified Ruby program. This metric includes the lines of all methods and annotations that were used to verify the method in question. For each verified method, RTR generates a separate Rosette program. We give the sizes of these resulting programs in the **Rosette LoC** column. Unsurprisingly, the LoC of the Rosette program increases with the size of the source Ruby program.

We present the median (**Time(s)**) and semi-interquartile range (**SIQR**) of the **Verification Time** required to verify all methods for an application over 11 runs. For each verified method, the **SIQR** was at most 2% of the verification time, indicating relatively little variance in the verification time. Overall, verification was fast, as might be expected for relatively small methods. The one exception was matrix multiplication. In this case, the slowdown was due to the extensive use of array operations mentioned above. We bounded array size (see § 4.4) at 10 for the evaluations. For symbolic arrays, this means Rosette must reason about every possible size of an array up to 10. This burden is exacerbated by matrix multiplication's use of two symbolic two-dimensional arrays.

Table 1. Method gives the class and name of the method verified. **Ru-LoC** and **Ro-LoC** give number of LoC for a Ruby method and the translated Rosette program. **Spec** is the number of method and variable type annotations we had to write. **Verification Time** is the median and semi-interquartile range of the time in seconds over 11 runs. **App Total** rows list the totals for an app, without double counting the same specs.

Method	Ru-LoC	Ro-LoC	Spec	Verification Time	
				Time(s)	SIQR
Money					
Money::Arithmetic#-@	7	29	4	5.69	0.14
Money::Arithmetic#eql?	11	40	3	5.74	0.03
Money::Arithmetic#positive?	5	24	3	5.40	0.01
Money::Arithmetic#negative?	5	24	2	5.42	0.01
Money::Arithmetic#abs	5	30	4	5.49	0.01
Money::Arithmetic#zero?	5	26	2	5.38	0.02
Money::Arithmetic#nonzero?	5	24	2	5.43	0.03
App Total	**43**	**197**	**10**	**38.56**	**0.25**
BusinessTime					
ParsedTime#-	10	58	8	6.28	0.02
BusinessHours#initialize	5	26	2	5.36	0.04
BusinessHours#non_negative_hours?	5	26	2	5.4	0.01
Date#week	7	32	2	5.53	0.01
Date#quarter	5	28	2	5.47	0.00
Date#fiscal_month_offset	5	25	2	5.41	0.02
Date#fiscal_year_week	7	33	2	5.53	0.03
Date#fiscal_year_month	12	35	3	5.65	0.02
Date#fiscal_year_quarter	9	42	2	5.72	0.03
Date#fiscal_year	11	32	4	5.81	0.03
App Total	**76**	**337**	**24**	**56.15**	**0.20**
Unitwise					
Unitwise::Functional.to_cel	4	25	2	5.42	0.03
Unitwise::Functional.from_cel	4	25	2	5.44	0.03
Unitwise::Functional.to_degf	4	22	1	5.41	0.01
Unitwise::Functional.from_degf	4	27	2	5.44	0.02
Unitwise::Functional.to_degre	4	27	2	5.44	0.01
Unitwise::Functional.from_degre	4	27	2	5.42	0.01
App Total	**24**	**153**	**6**	**32.55**	**0.11**
Geokit					
Geokit::Bounds#center	7	31	4	5.4	0.02
Geokit::Bounds#crosses_meridian?	7	35	6	5.59	0.12
Geokit::Bounds#==	9	60	5	5.97	0.13
Geokit::GeoLoc#province	5	26	2	5.52	0.11
Geokit::GeoLoc#success?	5	26	2	5.51	0.05
Geokit::Polygon#contains?	26	68	10	10.8	0.07
App Total	**59**	**246**	**21**	**38.80**	**0.50**
Boxroom					
UserFile#move	12	34	3	5.57	0.05
Matrix					
Matrix.*	57	94	9	334.35	3.99
Total	**271**	**1061**	**73**	**505.98**	**5.10**

Finally, Table 1 lists the number of type **specifications** required to verify each method. These are comprised of method type annotations, including the refinement type annotations for the verified methods themselves, and variable type annotations for instance variables. Note that we do not quantify the number of type annotations used for Ruby's core and standard libraries, since these are included in RDL.

We observe that there is significant variation in the number of annotations required for each application. For example, `Unitwise` required 6 annotations to verify 6 methods, while `Geokit` required 21 annotations for 6 methods. The differences are due to code variations: To verify a method, the programmer needs to give a refinement type for the method plus a type for each instance variable used by the method and for each (non-standard/core library) method called by the method.

Case Study. Next we illustrate the RTR verification process by presenting the exact steps required to specify and check the properties of a method from an existing Ruby library. For this example, we chose to verify the << method of the `Aggregate` library [1], a Ruby library for aggregating and performing statistical computations over some numeric data. The method << takes one input, data, and adds it to the aggregate by updating (1) the minimum @min and maximum @max of the aggregate, (2) the count @count, sum @sum, and sum of squares @sum2 of the aggregate, and finally (3) the correct bucket in @buckets.

```
def <<(data)
  if 0 == @count
    @min = data ; @max = data
  else
    @max = data if data > @max ; @min = data if data < @min
  end
  @count += 1 ; @sum += data ; @sum2 += (data * data)
  @buckets[to_index(data)] += 1 unless outlier?(data)
end
```

We specify functional correctness of the method << by providing a refinement type specification that declares that after the method is executed, the input data will fall between @min and @max.

```
type :<<, '(Integer data) → Integer { @min≤data≤@max }', verify: :bind
```

Here, the symbol :bind is an arbitrary label. To verify the specification, we load the library and call the verifier with this label:

```
rdl_do_verify :bind
```

RTR proceeds with verification in three steps:

- first use RDL to type check the basic types of the method,
- then translate the method to Rosette (using the translation of § 4), and
- finally run the Rosette program to check the validity of the specification.

Initially, verification fails in the first step with the error

```
error : no type for instance variable '@count'
```

To fix this error, the user needs to provide the correct types for the instance variables using the following type annotations.

```
var_type :@count, 'Integer'
var_type :@min, :@max, :@sum, :@sum2, 'Float'
var_type :@buckets, 'Array<Integer>'
```

The << method also calls two methods that are not from Ruby's standard and core libraries: to_index, which takes a numeric input and determines the index of the bucket the input falls in, and outlier ?, which determines if the given data is an outlier based on provided specifications from the programmer. These methods are challenging to verify. For example, the to_index method makes use of non-linear arithmetic in the form of logarithms, and it includes a loop. Yet, neither of the calls to_index or outlier ? should affect verification of the specification of <<. So, it suffices to provide type annotations with a pure label, indicating we want to use uninterpreted functions to represent them:

```
type : outlier ?, '(Float i) → Bool b', :pure
type :to_index , '(Float i) → Integer out', :pure
```

Given these annotations, the verifier has enough information to prove the post-condition on <<, and it will return the following message to the user:

```
Aggregate instance method << is safe.
```

When verification fails, an unsafe message is provided, combined with a counterexample consisting of bindings to symbolic values that causes the postcondition to fail. For instance, if the programmer *incorrectly* specified that data is less than the @min, *i.e.,*

```
type :<<, '(Integer data) → Integer { data < @min }'
```

Then RTR would return the following message:

```
Aggregate instance method << is unsafe.
Counterexample: (model [real_data 0][real_@min 0] ...)
```

This gives a binding to symbolic values in the translated Rosette program which would cause the specification to fail. We only show the bindings relevant to the specification here: when real_data and real_@min, the symbolic values corresponding to data and @min respectively, are both 0, the specification fails.

6 Related Work

Verification for Ruby on Rails. Several prior systems can verify properties of Rails apps. *Space* [26] detects security bugs in Rails apps by using symbolic execution to generate a model of data exposures in the app and reporting a bug if the model does not match common access control patterns. Bocić and Bultan proposes *symbolic model extraction* [14], which extracts models from Rails apps at runtime, to handle metaprogramming. The generated models are then used to verify data integrity and access control properties. *Rubicon* [25] allows

programmers to write specifications using a domain-specific language that looks similar to Rails tests, but with the ability to quantify over objects, and then checks such specifications with bounded verification. *Rubyx* [15] likewise allows programmers to write their own specifications over Rails apps and uses symbolic execution to verify these specifications.

In contrast to RTR, all of these tools are specific to Rails and do not apply to general Ruby programs, and the first two systems do not allow programmers to specify their own properties to be verified.

Rosette. Rosette has been used to help establish the security and reliability of several real-world software systems. Pernsteiner et al. [27] use Rosette to build a verifier to study the safety of the software on a radiotherapy machine. *Bagpipe* [42] builds a verifier using Rosette to analyze the routing protocols used by Internet Service Providers (ISPs). These results show that Rosette can be applied in a variety of domains.

Types For Dynamic Languages. There have been a number of efforts to bring type systems to dynamic languages including Python [10, 12], Racket [36, 37], and JavaScript [11, 23, 35], among others. However, these systems do not support refinement types.

Some systems have been developed to introduce refinement types to scripting and dynamic languages. *Refined TypeScript* (RSC) [40] introduces refinement types to TypeScript [13, 29], a superset of JavaScript that includes optional static typing. RSC uses the framework of Liquid Types [31] to achieve refinement inference. Refinement types have been introduced [21] to Typed Racket as well. As far as we are aware, these systems do not support mixins or metaprogramming.

General Purpose Verification Dafny [22] is an object-oriented language with built-in constructs for high-level specification and verification. While it does not explicitly include refinement types, the ability to specify a method's type and pre- and postconditions effectively achieves the same level of expressiveness. Dafny also performs modular verification by using a method's pre- and postconditions and labels indicating its purity or arguments mutated, an approach RTR largely emulates. However, unlike Dafny, RTR leaves this modular treatment of methods as an option for the programmer. Furthermore, unlike RTR, Dafny does not include features such as mixins and metaprogramming.

7 Conclusion and Future Work

We formalized and implemented RTR, a refinement type checker for Ruby programs using assume-guarantee reasoning and the just-in-time checking technique of RDL. Verification at runtime naturally adjusts standard refinement types to handle Ruby's dynamic features, such as metaprogramming and mixins. To evaluate our technique, we used RTR to verify numeric properties on six commonly used Ruby and Ruby on Rails applications, by adding refinement type specifications to the existing method definitions. We found that verifying these methods

took a reasonable runtime and annotation burden, and thus we believe RTR is a promising first step towards bringing verification to Ruby.

Our work opens new directions for further Ruby verification. We plan to explore verification of purity and immutability labels, which are currently trusted by RTR. We also plan to develop refinement type inference by adapting Hindley-Milner and liquid typing [31] to the RDL setting, and by exploring whether Rosette's synthesis constructs could be used for refinement inference. We will also extend the expressiveness of RTR by adding support for loop invariants and dynamically defined instance variables, among other Ruby constructs. Finally, as Ruby is commonly used in the Ruby on Rails framework, we will extend RTR with modeling for web-specific constructs such as access control protocols and database operations to further support verification in the domain of web applications.

Acknowledgements

We thank Thomas Gilray and the anonymous reviewers for their feedback on earlier versions of this paper. This work is supported in part by NSF CCF-1319666, CCF-1518844, CCF-1618756, CCF-1651225, CNS-1518765, and DGE-1322106.

Bibliography

[1] Aggregate (2017), https://github.com/josephruscio/aggregate
[2] Boxroom (2017), https://github.com/mischa78/boxroom
[3] Businesstime (2017), https://github.com/bokmann/business_time/
[4] Geokit (2017), https://github.com/geokit/geokit
[5] Matrix (2017), https://github.com/ruby/matrix
[6] Money (2017), https://github.com/RubyMoney/money
[7] Unitwise (2017), https://github.com/joshwlewis/unitwise/
[8] Verified ruby apps (2017), https://raw.githubusercontent.com/mckaz/milod.kazerounian.github.io/master/static/VMCAI18/source.md
[9] Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.Y.: A relational logic for higher-order programs (2017)
[10] Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: Rpython: A step towards reconciling dynamically and statically typed oo languages. DLS (2007)
[11] Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. ECOOP (2005)
[12] Aycock, J.: Aggressive type inference. International Python Conference (2000)
[13] Bierman, G., Abadi, M., Torgersen, M.: Understanding typescript. ECOOP (2014)
[14] Bocić, I., Bultan, T.: Symbolic model extraction for web application verification. ICSE (2017)
[15] Chaudhuri, A., Foster, J.S.: Symbolic security analysis of ruby-on-rails web applications. CCS (2010)
[16] Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. ECOOP (1995)
[17] Foster, J., Ren, B., Strickland, S., Yu, A., Kazerounian, M.: RDL: Types, type checking, and contracts for Ruby (2017), https://github.com/plum-umd/rdl
[18] Freeman, T., Pfenning, F.: Refinement types for ML (1991)
[19] Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: Jsketch: Sketching for java. ESEC/FSE 2015 (2015)
[20] Jones, C.: Specification and design of (parallel) programs. IFIP Congress (1983)
[21] Kent, A.M., Kempe, D., Tobin-Hochstadt, S.: Occurrence typing modulo theories. PLDI (2016)
[22] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. LPAR (2010)
[23] Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: Tejas: Retrofitting type systems for javascript (2013)
[24] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. TACAS (2008)
[25] Near, J.P., Jackson, D.: Rubicon: Bounded verification of web applications. FSE '12 (2012)

[26] Near, J.P., Jackson, D.: Finding security bugs in web applications using a catalog of access control patterns. ICSE '16 (2016)
[27] Pernsteiner, S., Loncaric, C., Torlak, E., Tatlock, Z., Wang, X., Ernst, M.D., Jacky, J.: Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. CAV (2016)
[28] Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in f* (2017)
[29] Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & efficient gradual typing for typescript (2015)
[30] Ren, B.M., Foster, J.S.: Just-in-time static type checking for dynamic languages. PLDI (2016)
[31] Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. PLDI (2008)
[32] Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in pvs. IEEE Trans. Softw. Eng. (1998)
[33] Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. LFP '92 (1992)
[34] Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. PLDI (2013)
[35] Thiemann, P.: Towards a type system for analyzing javascript programs. ESOP (2005)
[36] Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. OOPSLA (2006)
[37] Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. POPL (2008)
[38] Torlak, E., Bodik, R.: Growing solver-aided languages with rosette. Onward! (2013)
[39] Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell (2014)
[40] Vekris, P., Cosman, B., Jhala, R.: Refinement types for typescript (2016)
[41] Vytiniotis, D., Peyton Jones, S., Claessen, K., Rosén, D.: Halo: Haskell to logic through denotational semantics. POPL (2013)
[42] Weitz, K., Woos, D., Torlak, E., Ernst, M.D., Krishnamurthy, A., Tatlock, Z.: Scalable verification of border gateway protocol configurations with an smt solver. OOPSLA (2016)
[43] Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. PLDI (1998)

Modular Analysis of Executables using On-Demand Heyting Completion

Julian Kranz[1] and Axel Simon[2]

[1] Technische Universität München, Garching b. München, Germany, julian.kranz@in.tum.de
[2] Google Inc., Mountain View, CA, USA, axelsimon@google.com

Abstract A function-modular analysis is presented that computes precise function summaries in the presence of pointers and indirect calls. Our approach computes several summaries for a function, each specialized to a particular input property. A call site combines the effect of several summaries, based on what properties hold. The key novelty is that the properties are tailored to the function being analyzed. Moreover, they are represented in a domain-agnostic way by using Herbrand terms with variables. Callers instantiate these variables, based on their state. For each variable instantiation, a new summary is computed. Since the computed summaries are exact with respect to the property, our fixpoint computation resembles the process of Heyting completion where a domain is iteratively refined to be complete wrt. the intersection with a property. Our approach combines the advantages of a modular analysis, such as scalability and context-sensitivity, with the ability to compute meaningful summaries for functions that call other functions via pointers that were passed as arguments. We illustrate our framework in the context of inferring indirect callees in x86 executables.

Keywords: executable analysis, modular analysis, domain refinement

1 Introduction

One challenge in static analysis is the sheer size of the input program. This is particularly true for the analysis of executables that have easily an order of magnitude more statements than the corresponding source program. One key to scalability is the treatment of functions: On the one hand, the highest precision needed to prove the absence of run-time errors [2] can be obtained by inlining functions at each call-site with the cost of increasing the code to be analyzed dramatically. On the other hand, the duplicate evaluation of code can be avoided by performing a context-insensitive analysis in which all calling contexts of a function are merged and the return state is propagated to all call sites. A context-sensitive analysis without duplicate evaluation of functions can be obtained by inferring an input/output relation for each function. These function summaries are then combined to obtain a solution to the whole program using a global fixpoint computation. This approach is known as modular analysis [6].

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 291–312, 2018.
https://doi.org/10.1007/978-3-319-73721-8_14

```
struct Parity {
  virtual bool IsEven() = 0;
  virtual bool IsOdd() = 0;
};
struct Even : public Parity {
  bool IsEven() { return true;}
  bool IsOdd() { return false;}
};
struct Odd : public Parity {
  bool IsEven() {
    even_call++; return false;}
  bool IsOdd() { return true;}
  int even_call = 0;
};
```

```
void CheckEven() {
  Even even;
  Check(&even);
}
void CheckOdd() {
  Odd odd;
  Check(&odd);
  assert(odd.even_call > 0);
}
void Check(Parity* parity) {
  assert(parity->IsEven()
      != parity->IsOdd());
}
```

Figure 1. The running example C++ program.

We illustrate the challenges of a modular analysis using the code in Fig. 1. Here, the tests `CheckEven` and `CheckOdd` rely on the helper function `Check` to test an invariant of the two sub-classes `Even` and `Odd`. In a modular analysis, the methods `Even::IsEven` and `Even::IsOdd` are summarized by their effect of returning a constant value. The `Odd::IsEven` method modifies the `even_call` field pointed-to by `this`. A summary for this method must therefore assume the existence of a memory region at `*this` containing an **`int`** field. A precise summary of this function can be expressed by $x' = x + 1$ where x, x' is the value of the field before, resp. after, the call. A more challenging task is the summary of `Check`. Invoking the virtual methods accessed through the `parity` pointer amounts to an indirect function call. Without knowing which functions can be dispatched to, a summary of this function would have to make worst case assumptions: the invoked function may modify any memory reachable from global variables or the `this` pointer. Without any additional information, a summary of a function f_i containing indirect calls provides little or no information.

One way to ensure that no precision loss occurs, even in the presence of higher-order functions, is to limit the precision of the analysis up front. Specifically, by using only abstract domains that are able to condense the effect of a function without loss of precision, it is possible to compute a summary of a function even if it takes other functions as parameters (examples are type inference for functional programs [20], groundness analysis in Prolog [14] and instances of the IFDS framework [16]). These so-called condensing domains [9] are too imprecise to distinguish function behaviors based on pointer aliasing and numeric properties.

One particular kind of condensing domains are those whose meet distributes over the join of the lattice, i.e. $s \sqcap (t \sqcup u) = (s \sqcap t) \sqcup (s \sqcap u)$. Giacobazzi and Scozzari propose Heyting completion to make an existing domain meet-distributive [10]. This process adds new elements to a domain and may thereby refine an abstract domain until it is isomorphic to the concrete domain (which is a set of states and thus forms a distributive lattice). Heyting completion is therefore not generally

practical. In this paper, we use Heyting completion on-demand, namely when the analysis of a function requires it to avoid a severe loss of precision. In particular, once a particular property p is identified for which we want to avoid the lossy approximation $\{p\} \sqcap (s \sqcup t) \sqsupset (\{p\} \sqcap s) \sqcup (\{p\} \sqcap t)$, we track a new abstract state $p \rightarrow (\{p\} \sqcap s) \sqcup (\{p\} \sqcap t)$ and postpone the computation of a state in which p does not hold until a call-site is encountered that requires it. Ultimately, a function is summarized by a table $[p_1 \mapsto \{p_1\} \sqcap s_1, \dots p_n \mapsto \{p_n\} \sqcap s_n]$ and a call site c applies this summary by computing $\bigsqcup_i c \sqcap \{p_i\} \sqcap s_i$. We present an analysis whose predicates p state that an input to a function is equal to a function address. For instance, analyzing `CheckEven` creates a summary s_E of `Check` and stores the mapping (`parity->vtable[0]` $= a_E) \mapsto s_E$ where a_E is the address of `Even::IsEven`. A second summary of `Check` is created for the call site in `CheckOdd`. A call site such as `Check(rand() ? new Odd() : new Even())` can thereafter be evaluated by instantiating the two summaries and without re-analyzing `Check`.

Given a function with the predicated summary $[p_1 \mapsto s_1, \dots p_n \mapsto s_n]$ and a call-site with state c, the question arises if the predicates cover the state described by c, i.e. if $\gamma(c) \subseteq \gamma(p_1) \cup \ldots \cup \gamma(p_n)$. If not, new predicates $p_{n+1}, \dots p_{n+k}$ must be identified and a new summary must be computed for each predicate. For instance, calling `Check` with a new sub-class `Mark` whose method `Mark::IsEven` has address a_M, the computation of a new summary s_M of `Check` is needed, giving the table entry (`parity->vtable[0]` $= a_M) \mapsto s_M$. The challenge here is how to observe when a new predicate is needed and how to obtain it. Our contribution to this end is to represent predicates as a Herbrand abstraction (uninterpreted terms with variables as placeholder for other terms) which gives the analyzer the flexibility to express cross-cutting properties from several abstract domains. By evaluating these predicates wrt. a call-site state, the variables in the predicates will be instantiated with values that make the predicate true. Each variable assignment of a predicate gives a ground (i.e. fully instantiated) Herbrand term. A summary of the function is computed for each ground Herbrand term.

In summary, we make the following contributions towards modular analysis:

- We apply Heyting completion [10] on-demand in order to make the summary of a function complete for some predicate. Predicates are created on-demand, namely when incompleteness would lead to an unusably imprecise result.
- We propose Herbrand abstractions to express symbolic predicates that functions postulate and that call-sites instantiate, thereby providing an abstract interface between the base analysis and the completion mechanism.
- We present an implementation of this framework using an inter-procedural control-flow-graph analysis that is able to resolve function calls in an x86 executable compiled from a higher-order functional language.

The remainder of this paper is organized as follows: The next section defines a collecting and abstract semantics for an imperative language with indirect function calls. Section 3 generalizes these semantics to one that relates function inputs to outputs. Section 4 enhances this abstract interpretation with the generation of Herbrand terms and presents how a fixpoint is obtained in a modular way. Section 5 discusses our implementation before Sect. 6 presents related work.

$$\begin{aligned} Prog &::= FDecl^* \\ FDecl &::= \texttt{ident}()\{Stmt^*\} \\ Lhs &::= \texttt{ident.field}\underline{(}\rightarrow\texttt{field}\underline{)?} \\ Expr &::= Lhs \mid Loc \end{aligned} \qquad \begin{aligned} Stmt ::= &\ Loc_S : br\ \underline{(}Expr : Loc_S;\underline{)?}\ Loc_S \\ \mid &\ Loc_S : Lhs = Expr \\ \mid &\ Loc_S : call\ Expr \\ \mid &\ Loc_S : return \end{aligned}$$

Figure 2. The abstract grammar of the analyzed program. $\underline{(}E\underline{)?}$ denotes zero or one E.

2 Preliminary Definitions

In this section we define a language with functions and define a collecting semantics for it. Let [] denote an empty map, $m := [k_1 \mapsto v_1, \ldots k_n \mapsto v_n]$ a map where n values can be looked up with $m[k_i] = v_i$, let $m \setminus k$ denote a map without a mapping for k and let $m[k \mapsto v]$ denote an update at k. Let $\mathrm{dom}(m)$ denote the keys in m. Let $Loc = Loc_S \uplus Loc_M$ be the set of memory locations of a program P that is partitioned into statement labels Loc_S and statically and dynamically allocated memory regions Loc_M. Define $Loc_F \subseteq Loc_S$ to be the set of function entry points which coincide with the first statement in each function. We assume a C-like language where a variable v is stored at address $\&v \in Loc_M$. Let $\sigma \in \Sigma : Loc_M \to (\mathcal{F} \to \mathbb{V})$ define the program state with $\sigma(m)$ being a field map of the memory at address $m \in Loc_M$. A field map takes field names $\mathcal{F}$ to their content $\mathbb{V}$ where $\mathbb{V} := Loc \cup \mathbb{Z}$ denotes numeric values and addresses. The ability to partition a memory region into fields allows our analysis to express that a function call only accesses some but not all fields of a memory region.

Figure 2 defines the grammar of $P \in \mathcal{L}(Prog)$. A function is a sequence of statements consisting of conditional jumps, assignments, function calls, and returns. Note that every statement is preceded by its unique address $l \in Loc_S$. The statement $Lhs = Expr$ updates the specified field of a memory or, via the optional C arrow notation, a field in the pointed-to memory region. For brevity, we write `even_call` for `Even::IsEven.this` $\rightarrow$ `even_call` $\in \mathcal{L}(Lhs)$ (where `Even::IsEven` is the method in Fig. 1). The concrete semantics of a statement takes an input program state $\sigma \in \Sigma$ and returns a tuple consisting of the output state and the location where execution continues. The individual rules are explained below.

$$[\![\cdot]\!]^\natural : \mathcal{L}(Stmt) \times \Sigma \to (Loc_S \times \Sigma)$$

$$[\![l_s\colon br\ e : l_t; l_f]\!]^\natural \sigma = \begin{cases} \langle l_t, \sigma\rangle & \text{if } [\![e]\!]^\natural_{Expr}\sigma = 0 \\ \langle l_f, \sigma\rangle & \text{otherwise} \end{cases} \tag{1}$$

$$[\![l_s\colon \texttt{m.f} = e]\!]^\natural \sigma = \langle next(l_s), \sigma[m \mapsto \sigma(m)[f \mapsto v]]\rangle \text{ where } v = [\![e]\!]^\natural_{Expr}\sigma \tag{2}$$

$$[\![l_s\colon \texttt{m.f} \rightarrow \texttt{f}' = e]\!]^\natural \sigma = [\![l_s\colon \texttt{m}'.\texttt{f}' = e]\!]^\natural \sigma \text{ where } \&m' = [\![\texttt{m.f}]\!]^\natural_{Expr}\sigma \tag{3}$$

$$[\![l_s\colon call\ e]\!]^\natural \sigma = \langle \&\text{f}, \sigma[\texttt{f} \mapsto [\texttt{ret} \mapsto next(l_s)]]\rangle \text{ where } \&\texttt{f} = [\![e]\!]^\natural_{Expr}\sigma \tag{4}$$

$$[\![l_s\colon return]\!]^\natural \sigma = \langle l_r, \sigma \setminus \texttt{f}\rangle \text{ where } l_r = [\![\texttt{f.ret}]\!]^\natural_{Expr}\sigma \tag{5}$$

The evaluation of an expression $e \in \mathcal{L}(Expr)$ is defined as follows:

$$\llbracket\cdot\rrbracket^{\natural}_{Expr} \;:\; \mathcal{L}(Expr) \times \Sigma \to \mathbb{V}$$

$$\llbracket \texttt{m.f} \rrbracket^{\natural}_{Expr}\sigma = \sigma(m)(f) \tag{6}$$

$$\llbracket \texttt{m.f} \to \texttt{f}' \rrbracket^{\natural}_{Expr}\sigma = \llbracket \texttt{m}'.\texttt{f}' \rrbracket^{\natural}_{Expr}\sigma \text{ where } \&m' = \llbracket \texttt{m.f} \rrbracket^{\natural}_{Expr}\sigma \tag{7}$$

$$\llbracket l \rrbracket^{\natural}_{Expr}\sigma = l \qquad\qquad (\text{rule } Expr ::= Loc) \tag{8}$$

Jumps, defined by Eqn. 1, are unconditional if $e\ : l_t$ is omitted. Equation 2 updates the field f in $\sigma(m)$. It returns the program location following this statement using a function $next : Loc_S \to Loc_S$ that we assume to be suitably defined for all non-branching statements. A write through a pointer in Eqn. 3 assumes that the pointer value `m.f` matches the beginning of a memory region m' and is undefined otherwise. Thus, we do not model general pointer arithmetic and array accesses but assume that `parity->vtable[0]` is interpreted such that `vtable[0]` is a field name. Our implementation supports general pointer arithmetic.

The call instruction in Eqn. 4 continues execution at the called function. It also creates a memory region with the same name as the function. This memory region serves as the stack frame. The return instruction in Eqn. 5 jumps to the location in the local variable `f.ret`, where `f` is the current function. (Note that supporting recursion requires the use of unique names for stack frames as done in the implementation.) Moreover, we assume that function arguments are copied by the caller into the stack frame of the callee. The semantics of expressions in Eqns. 6 to 8 is straightforward.

A suitable collecting semantics is the classic merge-over-all-path solution. Let $\Sigma_s \subseteq \Sigma$ be the initial state at the program entry point l_{main}. We define:

Definition 1. *The collecting semantics of P is a map $col_P : Loc_S \to \wp(\Sigma)$ satisfying $\Sigma_s \subseteq col_P(l_{main})$ and for all $l : stmt \in P$, $\sigma \in col_P(l)$, and $\langle \sigma', l' \rangle = \llbracket l\text{: } stmt \rrbracket^{\natural}(\sigma)$ it holds that $\sigma' \in col_P(l')$.*

The structure $\langle Loc_S \to \wp(\Sigma), \dot{\subseteq}, \dot{\cup} \rangle$ is the cpo of the concrete domain where $\dot{\subseteq}$ and $\dot{\cup}$ are the point-wise liftings of the corresponding operations on the images of the map. The next section details how it is approximated by an abstract domain.

2.1 Abstract Interpretation of the Collecting Semantics

The segregation of memory into distinct regions lies at the heart of a modular analysis where a function summary leaves all but a small set of memory regions untouched. We therefore lift the concept of a memory region to the abstract.

Specifically, an abstract interpretation of the collecting semantics abstracts the unbounded set of memory regions in the concrete environments Σ by a bounded set of abstract memory regions $\mathcal{M}$. The memory regions define a set of non-overlapping areas of memory. The structure of a memory region $r \in \mathcal{M}$ is defined by a map $MS = \mathcal{M} \to (\mathcal{F} \to \mathcal{X})$ whose mappings are written $[r_1 \mapsto \phi^{\sharp}_1, \ldots, r_n \mapsto \phi^{\sharp}_n]$ where each $\phi^{\sharp}_i$ maps fields of a memory region r_i to a value domain variable $x \in \mathcal{X}$ that takes on values in $\mathbb{V} = \mathbb{Z} \cup Loc$.

The values of $X \subseteq \mathcal{X}$ are given by a domain $\mathcal{D}_X = \langle D_X, \sqsubseteq_{D_X}, \sqcup_{D_X}, \sqcap_{D_X}, \bot_D \rangle$. Here, X is the support set of $\mathcal{D}_X$, that is, the variables that D_X restricts. In our implementation, $\mathcal{D}_X$ is a reduced product [4] of several abstract domains. Since the inference of summaries requires the ability to express relations between input and output variables, a domain $d \in D_X$ must be concretized in a way that retains these relations. Thus, the concretization $\gamma_{\mathcal{D}_\mathcal{X}} : D_X \to \wp(\mathbb{V}^*)$ maps $d \in D_X$ to $\gamma_{\mathcal{D}_X}(d) = \{\boldsymbol{v}_1, \ldots\}$ where each vector $\boldsymbol{v}_i$ has one dimension for each abstract variable $x \in \mathcal{X}$. For instance, let $d \in D_{\{x,y\}}$ have its variables restricted by the interval constraint $x \in [3, 5]$ and the equality $x + 1 = y$ then $\langle x, y\rangle \in \gamma_\mathcal{D}(d) = \{\langle 3, 4\rangle, \langle 4, 5\rangle, \langle 5, 6\rangle\}$. We write $\boldsymbol{v}(x)$ to extract the value from the vector corresponding to the dimension $x \in \mathcal{X}$. Changes to the support set X of a domain $\mathcal{D}_X$ are implemented by two functions $\mathit{addVar}_x : \mathcal{D}_X \to \mathcal{D}_{X\cup\{x\}}$ (leaving x unrestricted) and $\mathit{delVar}_x : \mathcal{D}_{X\cup\{x\}} \to \mathcal{D}_X$ that are defined iff $x \notin X$.

Combining Memory Structure and Value Domain We now describe how MS and $\mathcal{D}_X$ are combined. For the sake of this section, let $\mathit{vars}(\mathit{ms}) \subseteq \mathcal{X}$ denote the variables occurring in $\mathit{ms} \in \mathit{MS}$. The lattice of our analysis contains elements $\langle m, d\rangle \in \mathit{MS} \times \{\mathcal{D}_X \mid X \subseteq \mathcal{X}\}$ such that $d \in \mathcal{D}_{\mathit{vars}(m)}$. We denote this universe as $\mathit{MS} \times D$. The concretization of $\mathit{MS} \times D$ to environments Σ proceeds in three steps: First, we define a function *embed* that updates an environment $\sigma \in \Sigma$ with the values in a vector $\boldsymbol{v} \in \mathbb{V}^*$ based on the fields of a memory region. The function recursively processes each mapping by pattern matching against the empty map and a map $\{r \mapsto \phi^\sharp\} \uplus m$ containing a mapping for region r and other mappings m:

$$\begin{aligned}
&\mathit{embed} : \mathit{MS} \times (\mathit{Loc} \cup \mathbb{Z})^* \times \Sigma \to \Sigma\\
&\mathit{embed}([], \boldsymbol{v}, \sigma) = \sigma\\
&\mathit{embed}([r \mapsto \phi^\sharp] \uplus m, \boldsymbol{v}, \sigma) = \mathit{embed}(m, \boldsymbol{v}, \sigma[r \mapsto \mathit{embed}_\phi(\sigma(r), \phi^\sharp, \boldsymbol{v})])\\
&\text{where } \mathit{embed}_\phi(m, \phi^\sharp, \boldsymbol{v}) = m[f \mapsto \boldsymbol{v}(\phi^\sharp(f)) \mid f \in \mathrm{dom}(\phi^\sharp)]
\end{aligned}$$

Second, we apply *embed* to the set of all concrete stores Σ, thereby obtaining $\{\mathit{embed}(m, \boldsymbol{v}, \sigma) \mid \sigma \in \Sigma\}$, the set of concrete stores in which the fields tracked by the abstract domain are restricted to values in $\boldsymbol{v}$. The final step is to compute this set for each value vector, giving the concretization function:

$$\begin{aligned}
\gamma_{\mathit{MS}\times D} &: \mathit{MS} \times D \to \wp(\Sigma)\\
\gamma_{\mathit{MS}\times D}(\langle m, d\rangle) &= \bigcup_{\boldsymbol{v}\in\gamma_\mathcal{D}(d)} \{\mathit{embed}(m, \boldsymbol{v}, \sigma) \mid \sigma \in \Sigma\}
\end{aligned}$$

We now address the task of defining the lattice operations on $\mathit{MS} \times D$. The problem to address is that two structures $m_1, m_2 \in \mathit{MS}$, that are propagated to the same program point, are associated with domains $d_i \in \mathcal{D}_{\mathit{vars}(m_i)}$, $i = 1, 2$, so that d_1 and d_2 range over different variables and cannot be compared or joined.

We address this problem using a cofibered abstract domains [22] and define three sound morphisms[3] $\mathit{addRegion}_r, \mathit{addField}_{r,f}, \mathit{renameField}_f : \mathit{MS} \times D \to$

[3] In categorical terms, $\mathit{MS} \times D$ is a Grothendiek construction $F \rtimes C$ using functor $F : C \to \mathbf{Cat}$ where C is a small category with $\mathit{obj}(C) = \mathit{MS}$ and $\mathbf{Cat}$ is a category

$MS \times D$ that are applied if the memory structures ms_1, ms_2 differ:

$$\langle m, d\rangle \xrightarrow{addRegion_r} \langle m[r \mapsto []], d\rangle \quad (9)$$

$$\langle [r \mapsto \phi^\sharp] \uplus m, d\rangle \xrightarrow{addField_{r,f}} \langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m, addVar_x(d)\rangle \quad (10)$$

$$\langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m, d\rangle \xrightarrow{renameField_{f,x,y}} \langle r \mapsto \phi^\sharp[f \mapsto y]] \uplus m, delVar_x([\![y := x]\!]^\sharp addVar_y(d))\rangle \quad (11)$$

Here, $[\![y := x]\!]^\sharp$ in Eqn. 11 is the update transformer on $\mathcal{D}_X$. By applying a composition of the three morphisms on the domain tuples $\langle m_i, d_i\rangle$, $i = 1, 2$, one can obtain tuples $\langle m_i', d_i'\rangle$ with $m_1' = m_2'$ so that the lattice operations $\sqsubseteq_{D_X}, \sqcup_{D_X}$ can be applied to d_i'. The morphisms can be shown as sound wrt. $\gamma_{MS\times D}$ and we obtain the abstract lattice $\langle MS \times D, \sqsubseteq_{MS\times D}, \sqcup_{MS\times D}, \bot_{MS\times D}\rangle$.

Example 1. We give an intuition on where the above morphisms are applied using an alias domain with universe $D_X = X \to \wp(Loc_M \cup \{a_{bad}\})$. It implements $addVar_x$ adding the mapping $x \mapsto \{a_{bad}\}$ where a_{bad} is a symbolic constant that represents all illegal addresses. Consider the following two functions:

```
void foo() {struct { int* a; } s;  if (rand()) s.a = &f; }
void bar() {struct{int*a;} s; if(rand()) s.a=&f; else s.a=&g;}
```

Assume that `s` initially points to a region without fields, i.e. $s \mapsto []$. Assume further that, in `foo` and `bar`, the then-branch updates `s` such that $s \mapsto [a \mapsto x_1]$. For `foo`, we have to apply the $addField_{s,a}$ morphism on the else-branch state before the join; the join, consequently, results in the alias set $x_1 \mapsto \{a_{bad}$, `&f`$\}$. In the else-branch of bar `bar`, the update creates, e.g., $s \mapsto [a \mapsto x_2]$. In this case, we have to apply $renameField_{a,x_2,x_1}$ so that the states to be joined have the same support set. The join results in $x_1 \mapsto \{$`&f`, `&g`$\}$ for the field `a`.

The presented memory structures MS do not allow for summarized memory regions as every abstract memory region $r \in \mathcal{M}$ corresponds to exactly one concrete memory region in σ, albeit at varying addresses. Although this suffices to illustrate our modular analysis, our implementation requires a simple form of summaries in form of weak updates. A concretization that caters for summarized memory regions [19] would complicate the presentation unnecessarily.

of small categories with $obj(\mathbf{Cat}) = \{\langle \mathcal{D}_X, \rho\rangle \mid X \subseteq \mathcal{X}, \rho : X \to (Loc_M \times \mathcal{F})\}$. Here, the translation ρ provides information on how X relates to the field names of memory regions. F maps a category of memory structures to a category of domains over variables in that memory structure. Thus, the category $F \rtimes C$ contains tuples $\langle m, \langle d, \rho\rangle\rangle \in obj(F \rtimes C)$ where $m \in MS$ and $d \in \mathcal{D}_{vars(m)}$. The morphisms $\langle m_1, \langle d_1, \rho_1\rangle\rangle \xrightarrow{(f,g)} \langle m_2, \langle d_2, \rho_2\rangle\rangle \in hom_{F\rtimes C}$ are pairs (f, g) where $m_1 \xrightarrow{f} m_2$ is a functor in C and g is a morphism $F(f)(\langle d_1, \rho_1\rangle) \xrightarrow{g} \langle d_2, \rho_2\rangle$ in $\mathbf{Cat}$. A morphism is sound if g defines an inclusion relation between its arguments [22] which is given if the values of d_1 are a subset of those in d_2 modulo the translation of variables: $g(\langle d_1, \rho_1\rangle, \langle d_2, \rho_2\rangle)$ iff $\forall \boldsymbol{v_1} \in \gamma_{\mathcal{D}_\mathcal{X}}(d_1). \exists \boldsymbol{v_2} \in \gamma_{\mathcal{D}_\mathcal{X}}(d_2). \forall x \in \mathrm{dom}(\rho_1) \wedge \rho_1(x) \in \mathrm{dom}(\rho_2^{-1})$. $\boldsymbol{v_1}(x) = \boldsymbol{v_2}(\rho_2^{-1}(\rho_1(x)))$. We omit ρ when defining morphisms as it is not needed.

3 Modular Program Semantics

In this section we generalize the collecting semantics and its abstract interpretation to function summaries. Specifically, we summarize the behavior of a function by a set of tuples $\langle \sigma, \bar{\sigma} \rangle$ that state how an input environment σ is mapped to an output environment $\bar{\sigma}$ and lift this relation to an abstract input/output relation.

We first define the input/output function semantics for a single input state. Recall that the semantics of calling f and returning from f in Eqns. 4 and 5 use the field `f.ret` to store the return address. In order to define the semantics of f independently of a caller, we evaluate it in an environment $\sigma = [\texttt{f.ret} \mapsto l_f^{res}]$ where $l_f^{res} \in Loc$ is a location that is not used in P.

Definition 2. *The semantics of f at $l_f \in Loc_S$ and executing in state σ is a map $col_f^\sigma : Loc_S \to \wp(\Sigma)$ satisfying $\sigma[\texttt{f.ret} \mapsto l_f^{res}] \subseteq col_f^\sigma(l_f)$ and for all $l : stmt \in P$, $\sigma' \in col_f^\sigma(l)$, and $\langle \sigma', l' \rangle = [\![l\colon stmt]\!]^\natural(\sigma')$ it holds that $\sigma' \in col_f^\sigma(l')$.*

We use the previous definition to define the relational semantics of f, that is, how each input state relates to the states at each statement of f:

Definition 3. *The relational semantics $rel_f : Loc_S \to \wp(\Sigma \times \Sigma)$ of a function f is given by $rel_f(l) = \{\langle \sigma, \bar{\sigma} \rangle \mid \sigma \in \Sigma \wedge \bar{\sigma} \in col_f^\sigma(l)\}$.*

Observe that rel_f is defined in terms of Equ. 4 which defines the semantics of a call to evaluate the called function rather than using the summary rel_f. We therefore use the following definition from now on:

$$[\![l_s\colon call\ e]\!]^\natural \sigma = \langle next(l_s), \bar{\sigma} \rangle \text{ where } \&\text{f} = [\![e]\!]^\natural_{Expr}\sigma \wedge \langle \sigma, \bar{\sigma} \rangle \in rel_f(l_f^{res}) \quad (12)$$

3.1 Abstract Interpretation of the Relational Semantics

The relational semantics of a function is approximated by an abstract domain $MS^2 \times D$ that is used to abstract $rel_f(l)$ for all locations $l \in Loc_S$ within function f. Here, $MS^2 = MS \times MS$ are two memory structures, the first describing the memory at the entry point of f, the second describing the memory at l. The relation between abstract and the concrete domain is given by $\gamma_{MS^2 \times D}$:

$$\gamma_{MS^2 \times D} : MS^2 \times D \to \wp(\Sigma \times \Sigma)$$
$$\gamma_{MS^2 \times D}(\langle m_{in}, m_{out}, d \rangle) = \bigcup_{\boldsymbol{v} \in \gamma_{\mathcal{D}}(d)} \{\langle embed(m_{in}, \boldsymbol{v}, \sigma), embed(m_{out}, \boldsymbol{v}, \sigma) \rangle \mid \sigma \in \Sigma\}$$

The concretization retains the relational character of rel_l in two ways: first, the *embed* functions are applied on the same numeric vector $\boldsymbol{v} \in \mathbb{Z}^*$ so that relational information between numeric variables are manifest in the concrete states. Second, the information of the abstract domain is embedded into the same $\sigma \in \Sigma$. As a consequence, a field in any concrete memory region in σ that is not present in either m_{in} nor m_{out} is not altered. These relational properties are illustrated in the following example:

Example 2. Let $l_e \in Loc_F$ be the entry point of method `Odd::IsEven()` in Fig. 1. The relational semantics at l_e is the identity, i.e. $rel_{l_e}(l_e) = \{\langle\sigma, \sigma\rangle \mid \sigma \in \Sigma\} = \gamma_{MS^2 \times D}(\langle m_{in}, m_{out}, d\rangle)$ where $m_{in} = m_{out} = []$ and $d \in \mathcal{D}_\emptyset$. Let $l_i \in Loc_S$ denote the location after the `even_call++` statement, then $\langle\sigma_{in}, \sigma_{out}\rangle \in rel_{l_e}(l_i)$ contains a memory region o at $l_o \in Loc_M$ that contains the object instance. An abstract state $s = \langle m_1, m_2, d\rangle \in MS^2 \times D$ with $rel_{l_e}(l_i) \in \gamma_{MS^2 \times D}(s)$ is $m_i = [\texttt{Odd::IsEven} \mapsto [\texttt{this} \mapsto y^i_{val}], o \mapsto [\texttt{even_call} \mapsto x^i_{val}]]$, $i = 1, 2$ and a value domain $d \in D$ containing the constraints $y^1_{val} = y^2_{val} = l_o$ and $x^1_{val} + 1 = x^2_{val}$.

The algebra $\langle MS^2 \times D, \sqsubseteq_{MS^2 \times D}, \sqcup_{MS^2 \times D}, \bot_{MS^2 \times D}, \bowtie_{MS^2 \times D}\rangle$ defines the abstract domain. Here, $\bowtie_{MS^2 \times D}$ is a special meet operator that combines the current state in a caller with the input state of a function summary. It is explained below. Other operations can be reduced to $\mathcal{D}$ using the following morphisms:

$$\langle m_1, m_2, d\rangle \overset{addRegion_r}{\longrightarrow} \langle m_1[r \mapsto []], m_2[r \mapsto []], d\rangle \tag{13}$$

$$\begin{aligned}\langle [r \mapsto \phi^\sharp_1] \uplus m_1, & \overset{addField_{r,f}}{\longrightarrow} \langle [r \mapsto \phi^\sharp_1[f \mapsto x_1]] \uplus m_1,\\ [r \mapsto \phi^\sharp_2] \uplus m_2, d\rangle & \qquad [r \mapsto \phi^\sharp_2[f \mapsto x_2]] \uplus m_2,\\ & \qquad [\![x_2 := x_1]\!]^\sharp addVar_{x_1}(addVar_{x_2}(d))\rangle\end{aligned} \tag{14}$$

$$\begin{aligned}\langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m_1, m_2, d\rangle \overset{renameField^1_f}{\longrightarrow} & \langle r \mapsto \phi^\sharp[f \mapsto y]] \uplus m_1, m_2,\\ & delVar_x([\![y := x]\!]^\sharp addVar_y(d))\rangle\end{aligned} \tag{15}$$

One obvious difference between these morphisms and those in Eqns. 9-11 is that they operate on two memory structures, namely the input ms_1 and the current state m_2 that eventually becomes the output state. For the sake of brevity, we do not handle cases where a function allocates new memory regions and can therefore assume that $\mathrm{dom}(m_1) = \mathrm{dom}(m_2)$ at all times. Under this assumption, we define Eqn. 13 and 14 that allow to add a region, resp., a field. The morphism $addField_{r,f}$ adds variables $x_1, x_2 \in \mathcal{X}$ that are made equal in the numeric domain, so that the domain maps each value of the field in the input to the same value in the output. Analogous to Eqn. 11, Eqn. 15 renames a field in m_1. We omit the symmetric definition $renameField^2_f$ that renames a field in m_2 for brevity.

3.2 Abstract Semantics of Memory Accesses

This section details the abstract semantics of memory accesses and illustrates how to deal with accesses to unknown locations. Figure 3 presents the abstract semantics for expressions (abstracting Eqns. 6 and 7 by Eqns. 16 and 17, respectively) and assignments (abstracting Eqns. 2 and 3 by Eqns. 18 and 19, respectively).

The expression semantics returns a set of variables or locations so that Eqn. 17 can return one variable for each dereferenced pointer. Note here that $\gamma_{\mathcal{D}}$ returns vectors of possible values and that $ms_2(m)(f)$ returns the domain variable that is used to index into the vector. Each element returned by the expression semantics is assigned by Eqn. 2 and the various results are joined. Equation 19 computes the assignment via a pointer as the join of writing to all possible locations $\&m'$.

$$
\begin{aligned}
[\![\cdot]\!]^\sharp_{Expr} &: \mathcal{L}(Expr) \times (MS^2 \times D) \to \wp(\mathcal{X} \cup Loc) \\
[\![\texttt{m.f}]\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle &= \{ms_2(m)(f)\} & (16) \\
[\![\texttt{m.f} \to \texttt{f}']\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle &= \bigcup_{\&m' \in \gamma_{\mathcal{D}}(d)(ms_2(m)(f))} [\![\texttt{m}'.\texttt{f}']\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle & (17) \\
[\![\cdot]\!]^\sharp &: \mathcal{L}(Stmt) \times (MS^2 \times D) \to Locs_S \times MS^2 \times D \\
[\![l_s\colon \texttt{m.f} = e]\!]^\sharp\langle ms_1, ms_2, d\rangle &= \langle next(l_s), & (18) \\
&\quad \bigsqcup_{e' \in [\![e]\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle} \langle ms_1, ms_2, [\![ms_2(m)(f) = e']\!]^\sharp d\rangle\rangle \\
[\![l_s\colon \texttt{m.f} \to \texttt{f}' := e]\!]^\sharp\langle ms_1, ms_2, d\rangle &= \langle next(l_s), & (19) \\
&\quad \bigsqcup_{\&m' \in \gamma_{\mathcal{D}}(d)(ms_2(m)(f))} [\![l_s\colon \texttt{m}'.\texttt{f}' = e]\!]^\sharp\langle ms_1, ms_2, d\rangle\rangle
\end{aligned}
$$

Figure 3. Abstract Expression Semantics.

Note that the expression $ms_2(m)(f)$ is undefined when either the memory region m does not exist in ms_2 or it does not contain a field f. Rather than handling this case in the semantic definition, we assume that the morphisms in Eqn. 13 and 14 are applied to prevent undefinedness. In case the transformer would access an unknown location through a pointer (i.e. `m.f` in Eqn. 17 or 19), a new region r is added using Eqn. 13 and `m.f` is restricted to point to it. Note that this behavior is not sound as it assumes that `m.f` does not alias with any other function inputs which may be wrong. We discuss this design choice in Sect. 5.

Example 3. We analyze `even_count++` in `Odd::IsEven` of Fig 1. Let `f.this` $== \&i$ be a test that forces `this` to point to the object instance i. For brevity, we use `f` for `Odd::IsEven`, `ev` for `even_count`, and write $d \in \mathcal{D}$ as set of constraints:

$$
\begin{aligned}
&\langle [f \mapsto []], [f \mapsto []], \emptyset\rangle \xrightarrow{addField_{f,this}} \langle [f \mapsto [this \mapsto x_1]], [f \mapsto [this \mapsto x_2]], \{x_1 = x_2\}\rangle \\
&\xrightarrow{addRegion_i} \xrightarrow{\texttt{f.this==}\&i} \langle [\ldots, i \mapsto []], [\ldots, i \mapsto []], \{x_1 = x_2 = l_i\}\rangle \xrightarrow{addField_{i,ev}} \\
&\langle [\ldots, i \mapsto [ev \mapsto x_3]], [\ldots, i \mapsto [ev \mapsto x_4]], \{x_1 = x_2 = l_i, x_3 = x_4\}\rangle \xrightarrow{[\![\texttt{f.this}\to\texttt{ev++}]\!]^\sharp} \\
&\langle [\ldots, i \mapsto [ev \mapsto x_3]], [\ldots, i \mapsto [ev \mapsto x_4]], \{x_1 = x_2 = l_i, x_3 + 1 = x_4\}\rangle
\end{aligned}
$$

The idea of applying morphisms as a precursor to a domain operation is also key to concisely define the $\bowtie_{MS^2 \times D}$ operation that combines a call-site state $\langle m^1_{in}, m^1_{out}, d_1\rangle$ with the summary of a function $\langle m^2_{in}, m^2_{out}, d_2\rangle$. Assuming that morphisms were applied so that m^1_{out}, the current state at the caller, and m^2_{in}, the input state of the callee summary, contain the same fields with the same variables while m^1_{in} and m^2_{out} share no variables, $\bowtie_{MS^2 \times D}$ reduces to $\sqcap_D$:

$$
\begin{aligned}
\langle m^1_{in}, m^1_{out}, d_1\rangle \bowtie_{MS^2 \times D} \langle m^2_{in}, m^2_{out}, d_2\rangle &= \langle m^1_{in}, m^2_{out}, d'\rangle \text{ where} \quad (20) \\
d' = delVar_{vars(m^1_{out}) \cup vars(m^2_{in})}(addVar_{vars(m^2_{out})}(d_1) &\sqcap_D addVar_{vars(m^1_{in})}(d_2))
\end{aligned}
$$

Here, *addVar* and *delVar* are used to add/remove a set of domain variables so that $\sqcap_D$ is applied to domains mapping variables of $m^1_{in}, m^1_{out}, m^2_{in}, m^2_{out}$ while

d' only contains variables relevant to the result. Ensuring that m^1_{out} and m^2_{in} contain the same fields amounts to matching the memory regions at the caller with those of the callee. Recall that the latter were created on-demand when computing the summary of the callee so that they have arbitrary names. We therefore compute a relation $R \subseteq \mathcal{M} \times \mathcal{M}$ between the caller's and the callee's memory regions by iteratively following pointers, starting with the actual and formal function arguments. If a pointer `s.p` can be followed to a region r in the callee but not in the caller, we apply $addRegion_r$, $addField_{s,p}$, to the caller and add $\langle r, r\rangle$ to R. For each pair added to R, we apply $addField_f$ in each region until both have the same fields. We appropriately name fields using $renameField_f$.

3.3 Computing Fixpoint of the Abstract Relational Semantics

This section details how the modular abstract semantics is used to compute a fixpoint of the whole program. A whole-program analysis populates a table $T \in \mathbb{T} = Loc_F \to MS^2 \times D$ that takes a function addresses to its summary. Since a function f may call other functions, a call statement in f will access T to obtain the most up-to-date summary for the called function. The semantics of the `call` statement is therefore parameterized by T:

$$[\![l_s\colon call\ e]\!]^\sharp_T \langle ms_1, ms_2, d\rangle = \langle next(l), \bigsqcup_{l_f \in \gamma_{\mathcal{D}}(d)([\![e]\!]^\sharp)} \langle ms_1, ms_2, d\rangle \bowtie_{MS^2\times D} T(l_f)\rangle \quad (21)$$

The resulting summary for f must therefore be re-computed if any summaries taken from T change. In the presence of recursive calls, widening [4] must be applied on the summaries to ensure termination.

The summary of f, given the table T and initial state s, is defined as follows:

Definition 4. *The abstract state of f is a map $abs^T_{f,s} : Loc_S \to MS^2 \times D$ with $s \sqsubseteq_{MS^2\times D} abs^T_f(l_f)$ and for all $l : stmt \in P$ and $\langle l', s'\rangle = [\![l : stmt]\!]^\sharp_T(abs^T_f(l))$ it holds that $s' \sqsubseteq_{MS^2\times D} abs^T_f(l')$. (Note: Eqns. 16-19 used $[\![l : stmt]\!]^\sharp \equiv [\![l : stmt]\!]^\sharp_T$.)*

Let $init = \langle m_1, m_2, \{x_1 = x_2 = l^{res}_f\}\rangle$ with $m_i = [f \mapsto [ret \mapsto x_i]]$, $i = 1, 2$ be the initial summary state. The summary semantics of f relates the first statement of the function at l_f with the location l^{res}_f that the return statement branches to:

Definition 5. *The abstract summary of f under T is $sum^T_f = abs^T_{f,init}(l^{res}_f)$.*

This concludes the presentation of the concrete relational semantics and the abstract summary domain and semantics. The next section tackles the challenge of computing precise summaries in the presence of indirect function calls.

4 On-Demand Heyting Completion

This section details how we use Herbrand terms to refine a function summary in cases where the most generic input would lead to an unacceptable precision loss. In particular, the next sections discuss the creation of Herbrand terms to express a need for refinement, the computation of a specialized function summary and the *call* semantics that combines specialized function summaries.

4.1 Extracting Refinement Information using Herbrand Terms

The challenge in specializing the summary of `Check` in Fig. 1 is that the variable over which to specialize is not known until the indirect call `parity->IsEven()` is analyzed. Our solution is that the analysis poses the question "What value can `parity->vtable[0]` take on?" to all callers of `Check` who may answer "The expression `parity->vtable[0]` may contain `&Odd::IsEven()`". (Recall that we use `vtable[0]` as a field name to fit our restricted grammar.) For each different answer, the summary of the analysis is specialized to the value in that answer. The analysis of `Check` can now proceed to the next indirect call `parity->IsOdd()` for which a new question is posed to the caller. Once the indirect function calls are resolved, `Check` can be summarized without posing further questions.

The "question" in the exposition above is represented by a Herbrand term that contains variables in places where the answer is expected. The answer to the question is given by a set of ground Herbrand terms, that is, Herbrand terms where the variables have been replaced by values.

Definition 6. *Herbrand terms $Herb = \mathcal{L}(Term)$ are defined by the grammar*

$$\begin{aligned} Term ::= &\ \texttt{constructor}\ Term^* \\ \mid &\ \texttt{variable} \end{aligned}$$

where `variable` *is drawn from $\mathcal{X}_H$. Note that $\mathcal{X}_H$ is distinct from $\mathcal{X}$. Let $vars(h)$ denote all variables in $h \in Herb$. Let $GHerb = \{h \in Herb \mid vars(h) = \emptyset\}$ denote ground Herbrand terms. A substitution $\theta \in \Theta : \mathcal{X}_H \to Herb$ is a total map with $\theta(x) = x$ except for a finite number of variables $y \in \mathcal{X}_H$ where $\theta(y) \neq y$. We write $[x/y] \in \Theta$ with $[x/y](x) = y$ and $[x/y](v) = v$ for all $v \neq x$. Given a term $h \in Herb$, we write $\theta(h)$ to denote the result of replacing all variables x in h by $\theta(x)$. Let $\theta(H) = \{\theta(h) \mid h \in H\}$ be the lifting to sets.*

The generic nature of Herbrand terms enables us to formulate questions that cut across several abstract domains in an abstract state $\langle m_1, m_2, d\rangle \in MS^2 \times D$.

Example 4. Suppose that the constructor *Deref* and *Field* are used by the memory domain $m \in MS$ to denote a pointer or field access, respectively, while *ConstPtr* is used by the numeric domain $d \in D$ to denote a function pointer. Then the term $ConstPtr\,(Field\,(Deref\,\texttt{parity})\ \texttt{vtable[0]})\ a_E$ is the request to access the field `vtable[0]` of the memory region pointed-to by `parity` and to extract the value as a constant pointer, denoting the result by $a_E \in \mathcal{X}_H$. This query accesses $m(f) = [\texttt{parity} \mapsto x, ...]$ where f is the frame of the currently analyzed function to obtain the numeric variable $x \in \mathcal{X}$ that contains the points-to set of `parity`. The numeric domain d is queried for the points-to set of x which resolves to, say, the address of memory region `even` $\in \mathcal{M}$. Finally, the memory domain is used to look up $m(\texttt{even}) = [\texttt{vtable[0]} \mapsto vt_E, \ldots]$ and d is queried for the values of vt_E, the constant address `vtable[0]` of `Even`, which becomes the solution of a_E.

For the sake of readability, we leave the exact definition of the term structure open and write `var->field`...`->field`$= a_E$, that is, we use C-like access paths

that generalize $\mathcal{L}(Expr)$ by allowing several indirections. Moreover, we also omit the memory region (i.e. we write `this->vtable[0]` instead of `f.this->vtable[0]`) since a Herbrand term is always relative to the stack frame of the current function.

Herbrand terms are used in the abstract semantics when a precise value is needed. For instance, the *call e* instruction requires a precise value for the function address e that determines which function is being invoked. An answer is computed using a function *herbEval* that evaluates a term set (e.g. $\{``e = x''\} \subseteq Herb$ for the call) given an abstract state. *herbEval* has the following signature:

$$herbEval\colon \wp(Herb) \times MS^2 \times D \to \wp(\Theta) \times \wp(Herb)$$

For variables $a_1, \ldots a_n$ in the input Herbrand terms, *herbEval* returns assignments in form of substitutions $\theta_1, \ldots \theta_k$ where each $\theta_j = [a_1/c_1^j, \ldots a_n/c_n^j]$ maps variables to constants $c_1^j, \ldots c_n^j \in \mathbb{V}$, $j = 1, \ldots k$, or it rewrites the Herbrand terms into terms over the function's input arguments. In order to illustrate this, we say that a Herbrand term h_i matches a domain variable x_i if h_i represents a field access (possibly via one or more pointer indirections) whose value is given by the domain variable x_i. We give an intuitive overview of *herbEval* by describing the four cases it distinguishes:

A set of values for tabulation can be constructed. The term h_i with variable a_i matches a domain variable x_i, $i = 1, \ldots n$. In case x_i are finite in the value domain state d, *herbEval* returns a set of constant value vectors $\boldsymbol{c}^1, \ldots \boldsymbol{c}^k \in \{\langle \boldsymbol{v}(x_1), \ldots \boldsymbol{v}(x_n)\rangle \mid \boldsymbol{v} \in \gamma_{\mathcal{D}_X}(d)\}$ in the form of k substitutions $\theta_j = [a_1/\boldsymbol{c}^j(x_1), \ldots a_n/\boldsymbol{c}^j(x_n)] \in \Theta$. For example, $herbEval(\{\mathtt{m}.f = a\}, \langle [\mathtt{m} \mapsto [f \mapsto x_1]], [\mathtt{m} \mapsto [f \mapsto x_2]], d\rangle)$ evaluates to $\langle \{[a/42]\}, \emptyset\rangle$ where $d = \{x_2 = 42\}$ represents the value domain.

An exact precondition can be synthesized. A term h_i matches a variable x_i. There exists $x_i' = x_i$ where x_i' is a domain variable of a field in the input memory region. For each x_i', we return a Herbrand term h_i' that matches x_i'. For example, $herbEval(\{\mathtt{m}.f = a\}, \langle [\mathtt{m} \mapsto [f \mapsto x_1], \mathtt{r} \mapsto [g \mapsto x_2]], [\mathtt{m} \mapsto [f \mapsto x_3], \mathtt{r} \mapsto [g \mapsto x_4]], d) = \langle \emptyset, \{\mathtt{r}.g = a\}\rangle$ if $d = \{x_2 = x_3\}$ is the value domain.

A sufficient precondition can be synthesized. The term h_i matches a variable x_i. There exist several variables $\{x_i^1, \ldots x_i^{k_i}\}$ from which there is a flow of information to x_i. We translate the single term h_i to Herbrand terms $h_i^1, \ldots h_i^{k_i}$ that match $x_i^1, \ldots x_i^{k_i}$ and return the term $\mathtt{Set}\ h_i^1 \ldots h_i^{k_i}$. For example, $herbEval(\{\mathtt{t}.q = a\}, [\mathtt{u} \mapsto [r \mapsto x_1], \mathtt{v} \mapsto [s \mapsto x_2]], [\ldots, \mathtt{t} \mapsto [q \mapsto x_3]], d) = \langle \emptyset, \{\mathtt{Set}\ \mathtt{u}.r = a_1\ \mathtt{v}.s = a_2\}\rangle$ where $d = [x_1 \mapsto \{\&p_1\}, x_2 \mapsto \{\&p_2\}, x_3 \mapsto \{\&p_1, \&p_2\}]$ represents the information of our aliasing domain $D_X = X \to \wp(Loc_M \cup \{a_{bad}\})$ used in Ex. 1. We will disregard this case until our discussion in Sect. 5.

No values can be synthesized. The term h_i matches no variable x_i nor can a field variable be added using $addField_{r,f}$. Thus, the values of variables in h_i are neither finite nor traceable to the input. An empty set of substitutions and Herbrand terms is returned. A warning is generated so that the analysis is sound if no warnings are emitted.

```
bool Case1() {
  Odd odd;
  Even even;
  Parity* parity =
    rnd() ? &odd : &even;
  return Check(parity);
}
```

```
void Case2(Parity *p) {
  Check(p);
}
void Case3(Parity *p,
                Parity *q) {
  Check(rnd() ? p : q);
}
```

Figure 4. Creating Herbrand terms for calls to `Check` in Fig. 1.

Example 5. We illustrate cases 1 to 3 using the functions in Fig. 4. We assume that `Check` has been analyzed with no specialization such that the first indirect call cannot be resolved. The resulting summary state is $\langle \bot_{MS^2 \times D}, H \rangle$ where $H = \{$`parity->vtable[0]` $= a\}$. As a consequence, H is evaluated at each call site using *herbEval*.

Consider the code of `Case1` in Fig. 4. When reaching the call to `Check` with summary state $s \in MS^2 \times D$, we evaluate *herbEval*(H, s) which amounts to evaluating the value of `parity->vtable[0]` in s. In this case, the state at the call site contains a finite set of values for this field, namely $\boldsymbol{v}_1 = \langle$`&Odd::IsEven`$\rangle$ and $\boldsymbol{v}_2 = \langle$`&Even::IsEven`$\rangle$. Thus, two new table entries have to be generated for `Check`, one for $H_1 = \{$`parity->vtable[0]` $=$ `&Odd::IsEven`$\}$ and $H_2 = \{$`parity->vtable[0]` $=$ `&Even::IsEven`$\}$. No further queries are raised. In `Case2`, the state at the call site of `Check` does not contain a finite set of values for the queried fields. However, there exists an equality relation with the parameter `p`. Thus, *herbEval* rewrites H to $H' = \{$`p->vtable[0]` $= a\}$ in terms of the parameter and propagates it to the callers of `Case2`. Finally, in `Case3`, *herbEval* is able to use the flow information computed by the points-to domain to determine that the l-values in `parity->vtable[0]` is a superset of the values in `p->vtable[0]` and `q->vtable[0]`. Thus, *herbEval* returns a single Herbrand term *Set* h_p h_q where $h_i \equiv \{i$`->vtable[0]` $= a_i\}$.

We omit a formal definition of *herbEval* as it is parametric in the value domain it operates on: In this case, *herbEval* extracts finite value sets and equalities between variables from the value domain, but other information can exploited as well. The next section discusses how *herbEval* is used to compute specialized summaries.

4.2 Specializing Summaries with Herbrand Terms

This section illustrates how a function summary is computed that is specialized wrt. a set of ground terms $H_g \in$ *GHerb*. To this end, we first define the lattice of an abstract domain where transformers can generate Herbrand terms whenever the function context needs to be refined. The lattice of this analysis is a product of $MS^2 \times D$ and a set of Herbrand terms *Herb* that we write as $\langle MS^2 \times D \times \wp(\mathit{Herb}), \sqsubseteq_H, \sqcup_H, \bot_H \rangle$. All lattice operations are the point-wise liftings, i.e. $\langle s_1, H_1 \rangle \sqsubseteq_H \langle s_2, H_2 \rangle \equiv s_1 \sqsubseteq_{MS^2 \times D} s_2 \wedge H_1 \subseteq H_2$, etc. In particular,

note that the product is not reduced [5], so that $\langle \bot_{MS^2\times D}, H\rangle \neq \bot_H$ unless $H = \emptyset$.

The analysis populates a table in $\mathbb{T}_{Herb} = Loc_F \times GHerb \to MS^2 \times D \times \wp(Herb)$. Each entry $\langle f, H_g\rangle \mapsto \langle s, H\rangle$ states that f, when specialized by H_g, has the summary s and requires further specializations by instantiating H in its callers. We define the following transformer to impose H_g on an abstract state:

$$[\![test\ H_g]\!]^\sharp : (MS^2 \times D) \to MS^2 \times D \quad (22)$$

For example, given the terms $H_g = \{\texttt{field} = 42\}$, the initial state $init$ in Sect. 3.3 is refined to $[\![test\ H_g]\!]^\sharp init = \langle m_1, m_2, \{x_1 = x_2 = l_f^{res}, x_3 = x_4 = 42\}\rangle$ where $m_i = [f \mapsto [ret \mapsto x_i, \texttt{field} \mapsto x_{i+2}]]$ for $i = 1, 2$. The semantics of a function f for a specialization H_g is defined by $sum_f^{T_H}$ that generalizes Def. 5:

Definition 7. *The specialized abstract summary of f under $T_H \in \mathbb{T}_{Herb}$ is given by $sum_f^{T_H} : GHerb \to (MS^2 \times D) \times \wp(Herb)$ where $sum_f^{T_H}(H_g) = abs^{T_H}_{f,[\![test\ H_g]\!]^\sharp init}$.*

Here, $T_H \in \mathbb{T}_{Herb}$ is the table of specialized summaries. Its elements are defined in terms of $sum_f^{T_H}$:

Definition 8. *$T_H \in \mathbb{T}_{Herb}$ is a well-formed table if $T_H(\langle f, H_g\rangle) = sum_f^{T_H}(H_g)$ for all $\langle f, H_g\rangle \in \mathrm{dom}(T_H)$.*

The analysis bootstraps by computing a summary for each function f with no specialization, thus providing the table entries with key $\langle f, \emptyset\rangle$. For any specialization H_g, a result $\langle s, H\rangle \in T_H(\langle f, H_g\rangle)$ may contain a non-empty set $H \in Herb$ that states how the function input must be specialized further so that the summary is an over-approximation of the function's concrete semantics. We now define how a call site of f instantiates H to a set of ground Herbrand terms $H_g \in GHerb$ that can be used to compute a specialized function summary $\langle f, H_g\rangle$ in T_H.

4.3 Combining Specialized Function Summaries

We now explain the differences between the semantics of the *call*-statement in Def. 21 and the following definition over the $(MS^2 \times D) \times \wp(Herb)$ domain:

$$[\![l_s\colon\ call\ e]\!]^\sharp_{T_H} \langle s, H\rangle = \langle \bot_{MS^2\times D}, H \cup H_f\rangle \sqcup_{MS^2\times D} \bigsqcup_{l_f \in \{l_f^1, \ldots l_f^n\}} applyEntries_{l_f}^{T_H}(s, \emptyset, \emptyset)$$

$$\langle \{[a/l_f^1], \ldots [a/l_f^n]\}, H_f\rangle = herbEval(\{``e = a"\}, s) \quad (23)$$

Rather than using the concretization function $\gamma_{MS^2\times D}$ to obtain the callee addresses l_f, we evaluate a Herbrand term $e = a$ in the current state $s \in MS^2 \times D$ where e is the called expression. We obtain a set of function addresses l_f^i, $i \in [1, n]$ and/or Herbrand terms H_f. Recall that a non-empty H_f are predicates over the inputs of this function that need to be restricted to a finite set of callers before

$$
\begin{aligned}
&\mathit{applyEntries}_f^{T_H} : ((MS^2 \times D) \times \wp(\mathit{Herb}) \times \wp(\mathit{GHerb})) \to (MS^2 \times D) \times \wp(\mathit{Herb}) \\
&\mathit{applyEntries}_f^{T_H}(s, H, H_g) = \\
&\quad \text{let } \langle s', H' \rangle \in T_H(\langle f, H_g \rangle) \text{ in} && (24)\\
&\quad \text{if } H' = \emptyset \text{ then } \langle s \bowtie_{MS^2 \times D} s', \emptyset \rangle \text{ else} && (25)\\
&\quad \text{let } \langle \Theta, H^{new} \rangle = \mathit{herbEval}(H \cup H', s) \text{ in} && (26)\\
&\quad \text{let } \overline{H'_g} = \{H'_g \mid H'_g = \theta(H \cup H') \cap \mathit{GHerb}, \theta \in \Theta, H_g \subseteq H'_g\} \text{ in} && (27)\\
&\quad \langle \bot_{MS^2 \times D}, H^{new} \rangle \sqcup_{MS^2 \times D} \bigsqcup_{H'_g \in \overline{H'_g}} \mathit{applyEntries}_f^{T_H}(s, H \cup H', H'_g) && (28)
\end{aligned}
$$

Figure 5. Applying a specialized function summary in $T_H \in \mathbb{T}_{Herb}$

this call has an effect. Thus, the predicates $H \cup H_f$ are returned with a bottom summary $\bot_{MS^2 \times D}$. The effect of each known callee at l_f^i is composed with the current state s using a helper function *applyEntries* that is defined in Fig. 5.

The idea of *applyEntries* is to find those specializations of callee f that match the caller state s and to combine those specializations with s. The arguments H and H_g always contain the same number of terms, where H_g is one specialization of H in s. In Eqn 24, we assume the table T_H contains an entry for the specialization $\langle f, H_g \rangle$. It is up to the fixpoint engine to compute a missing entry on-the-fly or to resume the evaluation of the caller once the entry is available. If the retrieved summary s' requires no new specializations, i.e. if $H' = \emptyset$, the summary s' is composed with the caller state in Eqn. 25 and returned. In case $H' \neq \emptyset$, the summary s' is an under-approximation and a more specialized summary must be consulted by instantiating $H \cup H'$ in the caller's state as done in Eqn. 26. The evaluation has two outcomes (which are not necessarily mutually exclusive): if $H^{new} \neq \emptyset$ then *herbEval* was able to translate the terms H' of the callee to inputs of the caller. These terms are therefore returned with the bottom summary $\bot_{MS^2 \times D}$ so that the caller will be refined. The second case is that $H \cup H'$ could be instantiated to concrete values in form of a set of substitutions Θ. Equation 27 applies Θ to obtain sets of ground terms $\overline{H'_g} \in \wp(\wp(\mathit{GHerb}))$ of which only those are returned that match the current specialization H_g. Each set $H'_g \in \overline{H'_g}$ is used to look up a more specialized summary of f by calling *applyEntries* recursively. We illustrate these definitions with an example.

Example 6. We illustrate the call semantics using the call to `Check` in `Case1` in Fig. 4. Assume that T_H has the following entries (`vt` is short for `parity->vtable`):

1	⟨`&Check`, $\emptyset$⟩	$\langle \bot_{MS^2 \times D}, \{$`vt[0]` $= a_0\}\rangle$
2	⟨`&Check`, {`vt[0]=&Even::IsEven`}⟩	$\langle s_1, \{$`vt[1]` $= a_1\}\rangle$
3	⟨`&Check`, {`vt[0]=&Odd::IsEven`}⟩	$\langle s_2, \{$`vt[1]` $= a_2\}\rangle$
4	⟨`&Check`, {`vt[0] = &Even::IsEven`, `vt[1] = &Even::IsOdd`}⟩	$\langle s_3, \emptyset \rangle$
5	⟨`&Check`, {`vt[0] = &Odd::IsEven`, `vt[1] = &Odd::IsOdd`}⟩	$\langle s_4, \emptyset \rangle$

The abstract call semantics in Eqn. 23 invokes $applyEntries^{T_H}_{\texttt{\&Check}}(s, \emptyset, \emptyset)$ where s is the caller state at the call site. The fact that Eqn. 24 returns a non-empty $H' = \{\texttt{vt[0]} = a_0\}$ means that a specialization needs to be computed, based on s which is done by Eqn. 26. Since s provides a finite set of values for a_0, $\Theta = \{[a_0/\texttt{\&Even::IsEven}], [a_0/\texttt{\&Odd::IsEven}]\}$ while H^{new} is empty. Applying these substitutions in Eqn. 27 gives two specializations in $\overline{H'_g}$, leading to two recursive calls in Eqn. 28, namely $applyEntries(s, \{\texttt{vt[0]} = a_0\}, \{\texttt{vt[0]} = \texttt{\&Even::IsEven}\})$ and $applyEntries(s, \{\texttt{vt[0]} = a_0\}, \{\texttt{vt[0]} = \texttt{\&Odd::IsEven}\})$. We only consider the first call as the second is analogous. Equation 24 extracts the 2nd table entry which, yet again, returns a non-empty H'. Equation 26 computes $\Theta = \{\theta_1, \theta_2\}$ where $\theta_1 = [a_0/\texttt{\&Even::IsEven}, a_1/\texttt{\&Even::IsOdd}]$, $\theta_2 = [a_0/\texttt{\&Odd::IsEven}, a_1/\texttt{\&Odd::IsOdd}]$ for the terms $H \cup H' = \{\texttt{vt[0]}= a_0, \texttt{vt[1]}= a_1\}$, thereby preserving the information at the call site that both, `vt[0]` and `vt[1]`, are taken from the same object instance. However, $\theta_2(H \cup H')$ is not a superset of H_g and is therefore discarded by Eqn. 27 as it is not a specialization of table entry 2. Thus, the only recursive call $applyEntries(s, H \cup H', \{\texttt{vt[0]} = \texttt{\&Even::IsEven}, \texttt{vt[1]} = \texttt{\&Even::IsOdd}\})$ consults table entry 4 and applies summary s_4 to the caller state using Eqn. 25.

4.4 Heyting Completion

In this section we show that the iterative tabulation of specialized function summaries is a Heyting completion, a well-known domain refinement technique [10]. A domain refinement adds new elements to an abstract domain. Our contribution is that completion is done on-demand, that is, only those elements are added to the lattice that are required by the program that is being analyzed.

Let $\langle L, \sqsubseteq_L, \sqcup_L, \sqcap_L\rangle$ be a complete lattice and $\alpha_X : L \to X$ a closure operator, i.e., monotone $Y \sqsubseteq_L Z \Rightarrow \alpha_X(Y) \sqsubseteq_L \alpha_X(Z)$, idempotent $\alpha_X(\alpha_X(Y)) = \alpha_X(Y)$, extensive $Y \sqsubseteq_L \alpha_X(Y)$, $\forall Y, Z \subseteq L$. Then $\langle L, \alpha, X, id\rangle$ is a Galois insertion [5].

Let $\Rightarrow \in L^2 \to L$ be a binary operator with $a \Rightarrow b = \bigsqcup_L \{c \in L \mid a \sqcap_L c \sqsubseteq_L b\}$. If $a \Rightarrow b \sqsubseteq_L b$ then $a \Rightarrow b$ is called the pseudo-complement of a relative to b. A lattice in which all pairs of elements have a pseudo-complement is called a Heyting algebra. We lift $\cdot \Rightarrow \cdot$ to sets $A, B \subseteq L$ as $A \Rightarrow B = \{a \Rightarrow b \in L \mid a \in A, b \in B\}$.

For any $X \subseteq L$ let $\curlywedge(X) = \{\bigsqcap_L Y \mid Y \subseteq X\}$ define the Moore closure of X. Let $A, B \in L$ such that α_A, α_B exist. Then the Heyting completion of A with respect to B is $\curlywedge(A \Rightarrow B)$. Let $\mathbb{H} = \bigcup_{H_G \subseteq GHerb} \{[\![test\ H_G]\!]^\sharp s \mid s \in MS^2 \times D\}$.

Theorem 1. *$\mathbb{H}$ is a Heyting completion of GHerb with respect to $MS^2 \times D$.*

Proof. First, show $\curlywedge(\mathbb{H}) = \mathbb{H}$. Let $[\![test\ H_i]\!]^\sharp s_i \in \mathbb{H}$ for $i = 1, 2$. Then $s := [\![test\ H_1]\!]^\sharp s_1 \sqcap_{MS^2\times D} [\![test\ H_2]\!]^\sharp s_2 = [\![test\ H_1]\!]^\sharp [\![test\ H_2]\!]^\sharp (s_1 \sqcap_{MS^2\times D} s_2) \in \mathbb{H}$ if there exists $H \in GHerb$ with $[\![test\ H]\!]^\sharp = [\![test\ H_1]\!]^\sharp \circ [\![test\ H_2]\!]^\sharp$. If "$e = c_i$" $\in H_i$ exists with $c_i \in \mathbb{Z}$ and $c_1 \neq c_2$ then $s = \bot_{MS^2\times D}$. Otherwise, since $s_1 \sqcap_{MS^2\times D} s_2$ has finitely many fields, there exists a finite $H \subseteq H_1 \cup H_2$. Thus, $H \in GHerb$.

Now show $\mathbb{H} = GHerb \Rightarrow MS^2 \times D$. Let $S_1 \sqsubseteq_{\mathbb{H}} S_2$ if for all $s_1 \in S_1$ there exists $s_2 \in S_2$ with $s_1 \sqsubseteq_{MS^2\times D} s_2$ and note that $\sqcap_{\mathbb{H}}$ exists due to $\curlywedge(\mathbb{H}) = \mathbb{H}$. Choose $H \subseteq GHerb$, $b \in MS^2 \times D$. Let $a = [\![test\ H]\!]^\sharp \langle [], [], \top_D\rangle \in MS^2 \times D$.

For the sake of contradiction, assume there exist c_i with $\{a\} \sqcap_{\mathbb{H}} c_i \sqsubseteq_{\mathbb{H}} \{b\}$ and $(\{a\} \sqcap_{\mathbb{H}} c_1) \sqcup_{\mathbb{H}} (\{a\} \sqcap_{\mathbb{H}} c_2) \not\sqsubseteq_{\mathbb{H}} \{b\}$. Let $C_i = \{[\![test\ H]\!]^\sharp s \mid s \in c_i\}$. Then $C_i \subseteq \mathbb{H}$ but also $C_1 \cup C_2 \subseteq \mathbb{H}$, thus $C_1 \cup C_2 \sqsubseteq_{\mathbb{H}} \{b\}$ which is a contradiction.

Corollary 1. *The entries of the table $T_H \in \mathbb{T}_{Herb}$ defined in Def. 8 are a partial Heyting completion of GHerb with respect to $MS^2 \times D$.*

Proof. By observing that $s \in \mathbb{H}$ for all $\langle f, H_g \rangle \mapsto \langle s, \emptyset \rangle \in T_H$.

5 Implementation and Evaluation

In this section, we discuss the implementation of our analyzer. Our analyzer reconstructs the control flow- and call graph of an x86 binary. The input binary is decoded and translated into the RReil language using the GDSL toolkit [12], starting at all function entry points defined in the ELF header.

Inter-procedurally, the analysis computes summaries for all functions starting from the initial state *init* defined in Sect. 3.3. The fixpoint computation proceeds by computing the summary of a callee before continuing at a call site using a dynamically updated partial order on the caller/callee relation. Intra-procedurally, the basic blocks of a function are discovered on-the-fly and we identify loops by observing jumps from higher to lower machine addresses. Within each loop, we apply a combined widening and narrowing operator for faster convergence [1].

The value domain D of the analysis is implemented as a set of three domains. The equality domain tracks predicates of the form $x = y + c$ for $x, y \in \mathcal{X}$ and $c \in \mathbb{Z}$. The pointer domain $D_X = X \to \wp(Loc_M \cup \{a_{bad}\}) \times X$ tracks relationships of the form $x_p - x_o \in \{l_1, \ldots, l_n\}$ with $x_p, x_o \in \mathcal{X}$, $l_i \in Loc_M$. Here, x_p is the pointer variable that is being tracked, x_o contains the offset relative to the beginning of l_i, the addresses of a memory region. Finally, the value set domain is used to track finite subsets of $\mathbb{Z}$ and intervals. We impose no fixed bound on the size of the subsets (i.e. no k-limiting) but widen a growing set to an interval. The three domains form a hierarchy where a parent domain forwards any domain operation to its child. For instance, the pointer domain transforms operations on pointer variables to operations on pointer offsets and passes them on to its child domain.

Section 4.1 raised the possibility that only necessary preconditions can be synthesized that are represented by a Herbrand term *Set* .... For instance, the call to `Case3Set` in Fig. 4 would generate the term `parity->vtable` $= a$ which is translated to a precondition *Set* $\{$`p->vtable` $= a_1$, `q->vtable` $= a_2\}$. In this case, instead of generating different table entries for each variable instantiation, we specialize $init_H$ with the join of all caller states, projected onto the variables in *Set* $\{\ldots\}$. When the caller state on these variables change, the function summaries need to be re-computed. Once necessary preconditions are generated, the analysis is no longer fully context-sensitive since the state at one caller can be propagated to the call site of a different caller. Our analysis also distinguishes summary memory regions that are created when accessing memory regions within loops. Any precondition generated in terms of summary memory regions are only necessary and never exact.

Binary	**Exact** H	**Set** H	**None**	**Avg.** H_G	**Indirect**	**Resolved**	**Time**	**Size**
`libgdsl_x86`	145	1	10	14.6	388	237	8.3h	1.1mb
`libgdsl_avr`	137	3	2	5.8	224	157	9.5m	303kb
`libgdsl_arm7`	82	0	4	3.4	153	96	13.4m	407kb
`echo`	0	0	0	N/A	7	2	134s	14.2kb
`cat`	0	0	0	N/A	7	2	8.2m	16.2kb

Table 1. Evaluation Results

Currently, each time a pointer is accessed that can be traced to the input, we create a fresh memory region. As a result, we implicitly assume that none of the pointer parameters alias. Thus, in general, our analysis is not sound. Future work will address how to incorporate the input aliasing configuration into the tabulation.

5.1 Evaluation

We have evaluated our implementation on the set of example binaries shown in Table 1. In particular, the benchmarks starting with `libgdsl` are libraries that are written in an ML-like functional programming language GDSL. GDSL is translated into idiomatic C code where higher-order functions are translated into C function pointers or heap-allocated closures containing function pointers [13].

Column **Exact** H contains the number of *call* / *br* statements that generated a Herbrand term with a single variable, i.e., terms that correspond to exactly one input memory field. The column **Set** H reports call sites that generate a term with a *Set* constructor, i.e. the cases where only necessary preconditions can be synthesized. The number of Herbrand terms that could not be translated to an input memory field is shown in **None**. Note that this number is low compared to the number of terms that can be translated to inputs, thereby showing that a summary abstract domain compares favorably against a backward analysis: the latter comes at the cost of implementing the backward semantic transformers. Column **Avg.** H_G contains the average number of instantiations for a Herbrand term. Columns **Indirect** and **Resolved** show the number of indirect *call*/*br* statements how many that were resolved to at least one target. Not all call sites can be resolved due to imprecision in our analysis as well as, for the libraries, due to the fact that many public functions take function pointers. Finally, columns **Time** and **Size** contain the analysis time and the size of the `.text` section.

Note that the gathering of the experimental data has been done using a preliminary prototype that tracks only a single summary per function by merging all requested function summaries. However, a more faithful implementation should only increase the precision of the analysis.

6 Related Work

One traditional approach of improving the precision of context-insensitive analysis is to only merge call sites whose last k parent call sites are the same (so-called

k-CFA) [18]. While the k-CFA approach improves the precision (i.e. Fig. 1 verifies with $k = 1$), it does so without consideration for the semantics of the program.

Modular analyses are context-sensitive by combining summaries of components/functions to a solution of the whole program. There are four principles [6]: compute a global fixpoint over some simplified semantics of each component, compute summaries under worst case assumptions, compute summaries using (possibly user-supplied) interfaces, and symbolic relational separate analysis (input/output abstractions). Most analyses combine some of these four principles.

Analyses that rely on condensing domains [9,14,15,16,20] perform a pure symbolic relational analysis based a restricted class of domains that comprise Herbrand terms with variables, Boolean functions and affine equalities.

The SeaHorn analyzer allows arguing over rich, numeric properties in a modular way [11]. It simplifies the input program into Horn clauses over predicates that are tailored to the analyzed program. These are then solved in a modular way. The downside is that no new invariants can be synthesized interprocedurally. Our tabulation over Herbrand terms is, in theory, less efficient than SeaHorn's Horn clauses since we store a summary state for each set of predicates. Yet, our summaries allow the computation of new invariants even interprocedurally.

Specializing the input of a summary falls into the category of summarizing with interfaces. One instance of this idea is the inference of preconditions that, when violated, lead to an error in the analyzed code [7]. An approach called "angelic verification" [8] goes further by restricting inputs to likely correct inputs.

Modular analyses that re-evaluate a component several times also adhere to the principle of computing summaries with interfaces, as each summary of a component is somehow specialized. The classic work on tabulation proposes to analyze a function for any possible input state and to combine table entries that match a call site [17]. Our approach is an on-demand tabulation that uses concrete values of function pointers as keys. Amato et al. perform tabulation based on the equality of the abstract input state [1]. Their tabulation approach may re-analyze a function unnecessarily, i.e. when a call site state has no match in the table but matches the join of several tabulated states. Moreover, matching tabulated states by equality may lead to non-monotone behavior [1, Example 1].

In the context of binary analysis, Xu et al. manually summarise functions using pre- and postconditions [23] that are similar to our Herbrand terms.

Finally, one "simplified semantics" idea is to break the program down so that it consists of parts that can be summarized with little precision loss (with the extreme of synthesizing transfer functions for groups of instructions [3,21]).

6.1 Conclusion

We presented a framework for modular analysis that judiciously computes multiple summaries. Each summary is specialized by Herbrand terms whose template is created by the function that is being analyzed and that is instantiated by its callees. We illustrated that this versatile approach corresponds to an on-demand Heyting completion of the domain and recovers indirect function calls.

References

1. G. Amato, F. Scozzari, H. Seidl, K. Apinis, and V. Vojdani. Efficiently intertwining widening and narrowing. Sci. Comput. Program., 120:1–24, 2016.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In Programming Language Design and Implementation, San Diego, California, USA, June 2003. ACM.
3. J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In R. Cousot and M. Martel, editors, Static Analysis Symposium, volume 6337 of LNCS, pages 182–196. Springer, September 2010.
4. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Principles of Programming Languages, pages 238–252, Los Angeles, California, USA, January 1977. ACM.
5. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In Principles of Programming Languages, pages 269–282, San Antonio, Texas, USA, January 1979. ACM.
6. P. Cousot and R. Cousot. Modular Static Program Analysis. In R. N. Horspool, editor, Compiler Construction, pages 159–178, Grenoble, France, April 2002. Springer. invited paper.
7. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic Inference of Necessary Preconditions, pages 128–148. Springer Berlin Heidelberg, Rome, Italy, 2013.
8. A. Das, S. Lahiri, A. Lal, and Y. Li. Angelic Verification: Precise Verification Modulo Unknowns, pages 324–342. Springer International Publishing, San Francisco, CA, USA, 2015.
9. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Domains Condensing. Trans. Comput. Log., 6(1):33–60, 2005.
10. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. Transactions on Programming Languages and Systems, 20(5):1067–1109, 1998.
11. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework, pages 343–361. Springer, San Francisco, California, USA, July 2015.
12. J. Kranz, A. Sepp, and A. Simon. GDSL: A Universal Toolkit for Giving Semantics to Machine Language. In C. Shan, editor, Asian Symposium on Programming Languages and Systems, Melbourne, Australia, December 2013. Springer.
13. J. Kranz and A. Simon. Structure-Preserving Compilation: Efficient Integration of Functional DSLs into Legacy Systems. In Principles and Practice of Declarative Programming. ACM, September 2014.
14. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. ACM Lett. Program. Lang. Syst., 2:181–196, March 1993.
15. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In Principles of Programming Languages, pages 330–341. ACM, January 2004.
16. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In Principles of Programming Languages, pages 49–61, San Francisco, California, USA, 1995. ACM.
17. M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

18. O. Shivers. Control-Flow Analysis of Higher-Order Languages. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.
19. H. Siegel and A. Simon. FESA: Fold- and Expand-based Shape Analysis. In Compiler Construction, volume 7791 of LNCS, pages 82–101, Rome, Italy, March 2013. Springer.
20. A. Simon. Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion. Science of Computer Programming, 95, Part 2(0):254–271, 2014.
21. A. Thakur and T. Reps. A Method for Symbolic Computation of Abstract Operations. In Computer Aided Verification, LNCS, pages 174–192, Berkeley, CA, 2012. Springer.
22. A. Venet. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In Static Analysis Symposium, LNCS, pages 366–382, London, UK, 1996. Springer.
23. Z. Xu, T. Reps, and B. Miller. Typestate Checking of Machine Code, pages 335–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

Learning to Complement Büchi Automata

Yong Li[1,2], Andrea Turrini[1], Lijun Zhang[1,2], and Sven Schewe[3]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] University of Liverpool, Liverpool, UK

Abstract. Complementing Büchi automata is an intriguing and intensively studied problem. Complementation suffers from a theoretical super-exponential complexity. From an applied point of view, however, there is no reason to assume that the target language is more complex than the source language. The chance that the smallest representation of a complement language is (much) smaller or (much) larger than the representation of its source should be the same; after all, complementing twice is an empty operation. With this insight, we study the use of learning for complementation. We use a recent learning approach for FDFAs, families of DFAs, that can be used to represent ω-regular languages, as a basis for our complementation technique. As a surprising result, it has proven beneficial not to learn an FDFA that represents the complement language of a Büchi automaton (or the language itself, as complementing FDFAs is cheap), but to use it as an intermediate construction in the learning cycle. While the FDFA is refined in every step, the target is an associated Büchi automaton that underestimates the language of a conjecture FDFA. We have implemented our approach and compared it on benchmarks against the algorithms provided in GOAL. The complement automata we produce for large Büchi automata are generally smaller, which makes them more valuable for applications like model checking. Our approach has also been faster in 98% of the cases. Finally we compare the advantages we gain by the novel techniques with advantages provided by the high level optimisations implemented in the state-of-the-art tool SPOT.

1 Introduction

The complementation of Büchi automata [15] is a classic problem that has been extensively studied for more than half a century; see [56] for a survey. The classic line of research on complementation has started with a proof on the existence of complementation algorithms [38,40] and continued to home in on the complexity of Büchi complementation, finally leading to matching upper [47] and lower [57] bounds ($\approx (0.76n)^n$) for complementing Büchi automata. This line of research has been extended to more general classes of automata, notably parity [49] and generalised Büchi [48] automata.

The complementation of Büchi automata is a valuable tool in formal verification (cf. [34]), in particular when a property that all runs of a model shall have is

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 313–335, 2018.
https://doi.org/10.1007/978-3-319-73721-8_15

provided as a Büchi automaton (one tests if the automaton that recognises the runs of a system has an empty intersection with the automaton that recognises the complement of the property language) and when studying language inclusion problems of ω-regular languages [3, 4, 52].

With the growing understanding of the worst case complexity, the practical cost of complementing Büchi automata has become a second line of research. In particular the GOAL tool suite [54] provides a platform for comparing the behaviour of different complementation techniques on various benchmarks [53].

Traditional complementation techniques use the automaton they seek to complement as a starting point for complex state space transformations. These transformations may lead to a super-exponential growth in the size. While this is generally unavoidable [57], we believe that there is no inherent reason to assume that the complement language is harder than the initial language; after all, complementing twice does not change the language[1]. This begs to ask the question, if we can—and if we should—avoid or reduce the dependency on the syntactic representation of the language we want to complement by a Büchi automaton.

This puts the focus on learning based approaches. The classic DFA learning algorithm L* has been proposed by Angluin in [6]. Based on L*, improvements and extensions have been made in [12,32,44]. They have been successfully applied in formal verification, for instance in compositional reasoning [17, 21, 23], system synthesis [2, 5, 13], and error localisation [20, 22]. Recently, Angluin's learning algorithm has been extended to ω-regular languages [8, 25, 35].

Families of DFAs [8, 35] (FDFAs), introduced in [7], have emerged as an excellent tool to represent ω-regular languages based on the representation of ultimately periodic words uv^ω as pairs (u, v). Based on the experience that DFAs tend not to be much larger than NFAs in practice, there is reasonable hope that FDFAs relate similarly to Büchi automata. Indeed, we have observed that, when we complement Büchi automata using existing determinisation techniques, it is often the case that their corresponding complement Büchi automata are much larger than themselves, while their complements by learning corresponding FDFAs have similar size to them, see Table 2 in Section 4. Moreover, FDFAs have proven to be well suited for learning [8, 35], which makes them an ideal starting point for developing a learning based automata complementation approach.

In a surprising twist, we found that FDFAs do not have to be learned to exploit them in a learning approach. Instead, we use candidate FDFAs $\mathcal{F}$ that are produced during the learning to infer Büchi automata $\underline{B}(\mathcal{F})$ that accept a subset of the ultimately periodic words represented by $\mathcal{F}$. Thus, while our learning algorithm is driven by a core that tries to learn a corresponding FDFA $\mathcal{F}$, it often terminates well before such an FDFA is found. This is possible, because the correspoinding FDFA is only a tool in the complementation algorithm. Broadly speaking, the algorithm uses a candidate $\mathcal{F}$, its complement $\mathcal{F}^{\mathsf{c}}$, and un-

[1] The typical model checking approach to complement the specification first also assumes that the translation into a Büchi automaton is equally efficient for the formula and its negation.

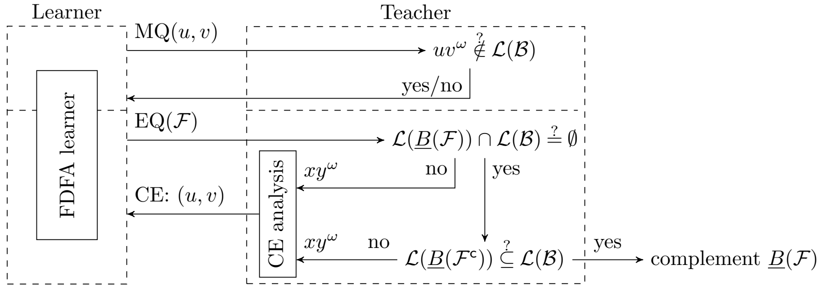


Fig. 1. The learning framework for complementing a Büchi automaton $\mathcal{B}$. The learner makes membership queries MQ(u, v), followed by the teacher revealing whether uv^ω is in the complement of $\mathcal{L}(\mathcal{B})$, and equivalence queries EQ$(\mathcal{F})$, upon which the teacher either replies that $\underline{B}(\mathcal{F})$ complements $\mathcal{B}$, or produces a counterexample CE: (u, v), such that the learner can refine $\mathcal{F}$ by either removing (u, v) from the language of $\mathcal{F}$ (if $uv^\omega \in \mathcal{L}(\mathcal{B})$), or by adding it to the language of $\mathcal{F}$ (otherwise).

derapproximations $\underline{B}(\mathcal{F})$ and $\underline{B}(\mathcal{F}^c)$ of their respective ω-languages as its main components, and it can stop as soon as $\underline{B}(\mathcal{F}^c)$ complements the given NBA.

This is feasible because complementing an FDFA $\mathcal{F}$ into an FDFA $\mathcal{F}^c$ is trivial (see Definition 5), and, while a Büchi automaton $\underline{B}(\mathcal{F})$ accepts only a subset of the ultimately periodic words defined by the FDFA $\mathcal{F}$ it under-approximates, we observe that the union of $\underline{B}(\mathcal{F})$ and $\underline{B}(\mathcal{F}^c)$ accepts all infinite words, which is justified by Proposition 2 in Section 3.3.

On first glance, this may sound as if this means that $\underline{B}(\mathcal{F})$ precisely captures the language represented by $\mathcal{F}$, but this is not always the case: an ultimately periodic word uv^ω has many representations as pairs, including e.g. (u, v), (uv, v), and (u, v^7), and it can happen that some are accepted by an FDFA $\mathcal{F}$, while others are accepted by its complement $\mathcal{F}^c$. In this case, we show that uv^ω will be accepted by $\underline{B}(\mathcal{F})$ *or* $\underline{B}(\mathcal{F}^c)$—and possibly by both of them (Proposition 2 in Section 3.3).

We use a variation of Angluin's classic DFA learning algorithm [6] to learn $\mathcal{F}$. Our learning approach, outlined in Figure 1, uses membership queries for $\mathcal{F}$ until a consistent automaton is created. It then turns to equivalence queries. For the membership queries, we use—cheap—standard queries [8, 35]. The novelty lies in a careful design of equivalence queries that make use of cheap operations whenever possible.

These equivalence queries are *not* executed with the FDFA $\mathcal{F}$ and its complement $\mathcal{F}^c$, but with the Büchi automata $\underline{B}(\mathcal{F})$ and $\underline{B}(\mathcal{F}^c)$ that underestimate them. We first check if $\underline{B}(\mathcal{F})$ has an empty language intersection with the automaton $\mathcal{B}$ we want to complement. This step is cheap, and if the answer is negative, then we get an ultimately periodic word uv^ω in the language of $\mathcal{B}$, where at least some representations of uv^ω are accepted by $\mathcal{F}$. We remove the representative provided by the teacher from the language of $\mathcal{F}$ and continue.

We then check if the language of $\underline{B}(\mathcal{F}^c)$ is included in the language of $\mathcal{B}$. This is an interesting twist, as language inclusion is one of the traditional justifications for complementing Büchi automata. But while the problem is PSPACE complete, it can usually be handled well by using efficient tools like RABIT [3, 4, 52]. Non-inclusion comes with a witness in the form of an ultimately periodic word uv^ω accepted by $\underline{B}(\mathcal{F}^c)$, but not by $\mathcal{B}$. Thus, some representations of uv^ω are (incorrectly) rejected by $\mathcal{F}$. We add them to the language of $\mathcal{F}$ and continue. Otherwise we have $\mathcal{L}(\underline{B}(\mathcal{F}^c)) \subseteq \mathcal{L}(\mathcal{B})$. We then conclude that $\mathcal{L}(\underline{B}(\mathcal{F})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$ and terminate the algorithm with $\underline{B}(\mathcal{F})$ as the complement of $\mathcal{B}$, which is justified by Theorem 2.

In a final bid for optimisation, we observe that this learning procedure can only terminate if $\underline{B}(\mathcal{F})$ and $\underline{B}(\mathcal{F}^c)$ are disjoint, which is justified by Corollary 2 in Section 3.4. If they are not, each ultimately periodic word uv^ω in their intersection will, in the final check, be a witness for language non-inclusion. It is, however, much cheaper to find. We therefore suggest that we check disjointness first and proceed to the more expensive language inclusion test only when the disjointness test fails.

Remark. We have also experimented with checking universality of $\mathcal{L}(\mathcal{B}) \cup \mathcal{L}(\underline{B}(\mathcal{F}))$ instead of checking language inclusion of $\underline{B}(\mathcal{F}^c)$ in $\mathcal{B}$ in the framework since this is a simple and more intuitive algorithm for complementing Büchi automata based on our learning framework. It has proven to be slower than the algorithm depicted in Figure 1 which confirms that our handling with the equivalence queries is more practical.

Contribution. The complementation of Büchi automata is a heavily researched field. However, to the best of our knowledge, all methods applied to it so far have been automata based. While this focus is natural, it is an important conceptional contribution to consider methods that do *not* focus on manipulating the automata we seek to complement.

Technically, publications about L*-style algorithms can be divided into two main classes: extensions of the L* family to new classes of automata [1,8,12,25,35, 36] and the works that provide suitable—and usually well-performing—teachers for a class of learning problems (e.g. [2,5,12,13,17–23,26,27,31,32,35,41,43,44]). This paper belongs to the latter class of contributions: we propose a simple and practical learning algorithm for complementing Büchi automata.

The performance of learning algorithms depends heavily on the implementation of the teacher. In line with other applications of L*-style algorithms, our contribution is the careful design of an FDFA teacher. In our context, membership queries are straight forward, and the challenge is exclusively in the equivalence queries. The PSPACE equivalence queries the teacher has to answer look like a show stopper. Adding the theoretical super-exponential blow-up incurred by complementing a Büchi automaton to the mix, it is like having the cards stacked against you.

Looking more closely at the challenges posed by equivalence queries, however, reveals that the high costs of equivalence checking can often be avoided. First and foremost, we can check if the candidate language intersects with the language of

the automaton we want to complement by a cheap emptiness query ($\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) \stackrel{?}{=} \emptyset$). When the emptiness holds, we check if $\mathcal{L}(\underline{B}(\mathcal{F}))$ and $\mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}}))$ intersect, using a second cheap emptiness query ($\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) \stackrel{?}{=} \emptyset$). The teacher only uses PSPACE-hard language inclusion queries ($\mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) \stackrel{?}{\subseteq} \mathcal{L}(\mathcal{B})$) once both previous queries are passed. These are usually few queries, and we found that, in spite of the theoretical complexity, existing tools can check inclusion sufficiently fast for this approach to be efficient.

We have implemented the learning-based approach in the tool Buechic based on ROLL [35]. Although we do not improve the theoretical complexity of complementing Büchi automata, our careful design of the FDFA teacher makes learning complement Büchi automata work reasonably well in practice. This is confirmed by the experiments we have performed on the roughly 500 Büchi automata from Büchi Store [55], the generated Büchi automata by SPOT for formulas in [50] and NCSB-Complementation [11].

In the performance evaluation, we were particularly interested in a comparison with GOAL [54]—considering the time to generate the complement automata and their size—as GOAL provides a comprehensive collection of the state-of-the-art techniques as well as a collection of benchmarks. It is therefore well suited for serving as a point of comparison with our novel technique.

In order to give a complete picture of the Büchi automata complementation state of the art, we have also compared Buechic against SPOT [24]. Differently from GOAL, SPOT only implements the most successful technique, and is a highly engineered state-of-the-art tool that has used the insight from GOAL and other automata manipulation techniques to obtain powerful heuristics for state space reduction on top of the principle techniques. While we consider the comparison with GOAL to be fair, comparing with SPOT is over-stretching what our tool can achieve—a bit like comparing a prototype for a new model checking approach with NuSMV. Moreover, SPOT takes advantage of a symbolic representation of the automata, by means of *Ordered Binary Decision Diagrams* (OBDDs) [14], while both Buechic and GOAL use an explicit graph data structure to represent the automata. This means that SPOT can work on multiple states and transitions simultaneously while Buechic and GOAL can only work on a single state/transition at a time. This is another reason why we consider the comparison of Buechic with GOAL to be fairer than with SPOT. Since SPOT does not provide a complementation function for general automata, but only for deterministic ones, we have derived one based on the implemented techniques (determinisation, complementation of deterministic automata, transformation to Büchi) to compare the advancement obtained by our technique with the advancement obtained by using symbolic encoding, states reduction, powerful heuristics, and performance optimisation.

The complement automata we produce are generally smaller for large Büchi automata than those generated by GOAL and SPOT, which makes them more valuable for applications like model checking. Moreover, Buechic has also been faster in 98% of the cases when compared to GOAL, though SPOT is often considerably faster due to its maturity and use of symbolic data structures.

Related Work. Current algorithms [15,28–30,33,38–40,42,45–47,51,53,54,56,57] for the complementation of Büchi automata are based on a *direct* complementation approach, which is quite different from learning. For a given Büchi automaton $\mathcal{B}$, these approaches use the *structure* of $\mathcal{B}$ as a base to construct a new Büchi automaton that recognises the complement language $\Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.

We use the learning algorithm instead to directly obtain an automaton that recognises $\Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$. It relies mainly on the *language* of $\mathcal{B}$ instead of on its structure. This allows for obtaining a small automaton for $\Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$, even one that is much smaller than $\mathcal{B}$.

Regarding the use of learning algorithms, there is a vast literature about regular languages (see, e.g., [2, 5, 6, 12, 13, 17, 20–23, 32, 44]); learning ω-regular languages [8, 25, 35] is a young and emerging field. In [25], they learn a Büchi automaton for an ω-regular language $\mathcal{L}$ by learning a DFA defined in [16]. The work proposed in [8] sets the general framework for learning ω-regular languages by means of FDFAs while [35] proposes a practical implementable framework by providing the appropriate FDFA teacher: it assumes that there exists an oracle for the target ω-regular language $\mathcal{L}$ and constructs an automaton accepting $\mathcal{L}$. In this paper we design the oracle for the FDFA teacher used in [35]; the oracle knows the complement of the language of $\mathcal{B}$ and is able to produce the appropriate counterexamples that are then analysed and returned to the learner.

Organisation of the Paper. After starting with some background and notation in Section 2, we describe our learning based complementation technique in Section 3. In Section 4, we evaluate our technique on standard complementation benchmarks and against the competitor algorithms from the GOAL suite and SPOT, before concluding the paper with Section 5.

2 Preliminaries

Let X and Y be two sets; we denote by $X \ominus Y$ their *symmetric difference*, i.e. the set $(X \setminus Y) \cup (Y \setminus X)$.

Let Σ be a finite set of *letters* called *alphabet*. A finite sequence of letters is called a (finite) *word*. An infinite sequence of letters is called an *ω-word*. We use $|\alpha|$ to denote the length of the finite word α and we denote by $\mathrm{last}(\alpha)$ the last letter of α. We use ε to represent an empty word. The set of all finite words on Σ is denoted by Σ^*, and the set of all ω-words is denoted by Σ^ω. Moreover, we also denote by Σ^+ the set $\Sigma^* \setminus \{\varepsilon\}$. Given a finite word $\alpha = a_0 a_1 \ldots$ and $i, k < |\alpha|$, we denote by $\alpha(i)$ the letter a_i and we use $\alpha[i : k]$ to denote the subword $\alpha' = a_i \ldots a_k$ of α, when $i \leq k$, and the empty word ε when $i > k$.

Definition 1. *A* nondeterministic Büchi automaton *(NBA) is a tuple $\mathcal{B} = (\Sigma, Q, I, \mathrm{T}, \mathrm{F})$, consisting of a finite alphabet Σ of input letters, a finite set Q of states with a non-empty subset $I \subseteq Q$ of initial states, a set $\mathrm{T} \subseteq Q \times \Sigma \times Q$ of transitions, and a set $\mathrm{F} \subseteq Q$ of accepting states.*

We denote the generic elements of an NBA $\mathcal{B}$ by Σ, Q, I, T, F, and we propagate primes and indices when necessary. Thus, for example, the NBA $\mathcal{B}'_i$ has states Q'_i, initial states I'_i, input letters Σ'_i, transition set T'_i, and accepting states F'_i; we use a similar notation for the other automata we introduce later.

An *run* of an NBA $\mathcal{B}$ over an ω-word $\alpha = a_0 a_1 a_2 \cdots \in \Sigma^\omega$ is an infinite sequence of states $\rho = q_0 q_1 q_2 \cdots \in Q^\omega$ such that $q_0 \in I$ and, for each $i \geq 0$, $\big(\rho(i), a_i, \rho(i{+}1)\big) \in \mathrm{T}$ where $\rho(i) = q_i$. A run ρ is *accepting* if it contains infinitely many accepting states, i.e. $\mathrm{Inf}(\rho) \cap \mathrm{F} \neq \emptyset$, where $\mathrm{Inf}(\rho) = \{\, q \in Q \mid \forall i \in \mathbb{N}.\exists j > i : \rho(j) = q \,\}$. A ω-word α is *accepted* by $\mathcal{B}$ if $\mathcal{B}$ has an accepting run on α, and the set of words $\mathcal{L}(\mathcal{B}) = \{\, \alpha \in \Sigma^\omega \mid \alpha \text{ is accepted by } \mathcal{B} \,\}$ accepted by $\mathcal{B}$ is called its *language*.

We call a subset of Σ^ω is an *ω-language* and the language of an NBA an *ω-regular language*. Words of the form uv^ω are called *ultimately periodic* words. We use a pair of finite words (u, v) to denote the ultimately periodic word $w = uv^\omega$. We also call (u, v) a *decomposition* of w. For an ω-language L, let $\mathrm{UP}(L) = \{\, uv^\omega \in L \mid u \in \Sigma^*, v \in \Sigma^+ \,\}$ be the set of all ultimately periodic words in L.

Theorem 1 (Ultimately Periodic Words [15]). *Let L, L' be two ω-regular languages. Then $L = L'$ if, and only if, $UP(L) = UP(L')$.*

An immediate consequence of the above theorem is that, for any two ω-regular languages L_1 and L_2, if $L_1 \neq L_2$ then there is an ultimately periodic word $xy^\omega \in \mathrm{UP}(L_1) \ominus \mathrm{UP}(L_2)$.

Definition 2. *A* deterministic finite automaton *(DFA) is a tuple $A = (\Sigma, Q, \bar{q}, \mathrm{T}, F)$, consisting of a finite alphabet Σ of input letters, a finite set Q of states with an initial state $\bar{q} \in Q$, a total transition function $\mathrm{T} \colon Q \times \Sigma \to Q$, and a set $F \subseteq Q$ of accepting (final) states.*

The complement A^{c} *of a DFA $A = (\Sigma, Q, \bar{q}, \mathrm{T}, F)$ is the DFA $A^{\mathsf{c}} = (\Sigma, Q, \bar{q}, \mathrm{T}, Q \setminus F)$.*

Given a DFA A and two states s and f, let $A^s_f = (\Sigma, Q, s, \mathrm{T}, \{f\})$ be the DFA obtained from A by setting its initial and accepting states to s and $\{f\}$, respectively.

A run of a DFA A over a word $\alpha = a_0 \cdots a_k \in \Sigma^*$ is a finite sequence of states $\rho = q_0 \cdots q_{k+1} \in Q^*$ such that $q_0 = \bar{q}$ and for every $0 \leq i \leq k$, $q_{i+1} = \mathrm{T}(q_i, a_i)$ where $k \geq 0$. The run ρ of A on α is accepting if $q_{k+1} \in F$. We denote by $\mathcal{L}(A)$ the language of A, i.e., the set of all words whose corresponding runs are accepted by A. We call the language of a DFA a *regular language*. Given an input word $\alpha \in \Sigma^*$ and the run ρ of A on α, we denote by $A(\alpha)$ the last reached state $\mathrm{last}(\rho)$. Given a DFA A with alphabet Σ, it holds that $\mathcal{L}(A^{\mathsf{c}}) = \Sigma^* \setminus \mathcal{L}(A)$.

Note that we require T to be total so to simplify the definitions in the remainder of the paper. Each DFA A with a non-total transition function can be transformed to a DFA A' as by Definition 2 such that $\mathcal{L}(A') = \mathcal{L}(A)$ by adding a fresh non-final sink state, and by letting T' agree with T where T is defined and mapping to this fresh sink state otherwise.

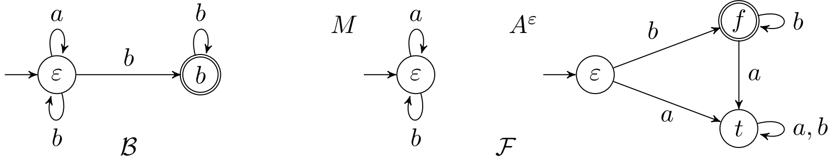


Fig. 2. An NBA $\mathcal{B}$ and an FDFA $\mathcal{F} = (M, \{A^\varepsilon\})$ recognising the same language $L = \Sigma^* \cdot b^\omega$.

Learning regular languages via DFAs was first proposed in [6], and the *right congruence* is the theoretical foundation for it to discover states in a regular language. A right congruence is an equivalence relation $\sim$ on Σ^* such that $x \sim y$ implies $xv \sim yv$ for every $x, y, v \in \Sigma^*$. We denote by $|\sim|$ the index of $\sim$, i.e. the number of equivalence classes of $\sim$. We use $\Sigma^*/_\sim$ to denote the equivalence classes of the right congruence $\sim$. A *finite right congruence* is a right congruence with a finite index. For a word $u \in \Sigma^*$, we denote by $[u]_\sim$ the class of $\sim$ in which u resides.

The main obstacle to learn ω-regular languages via Büchi automata is that there is a lack of right congruence for Büchi automata. Inspired by the work of Arnold [9], Maler and Stager [37] proposed the notion of *family of right-congruences*. Based on this, Angluin and Fisman [8] further proposed to learn ω-regular languages via a formalism called *family of DFAs*, in which every DFA corresponds to a right congruence.

Definition 3 (Family of DFAs [8]). *A* family of DFAs *(FDFA) over an alphabet Σ is a pair $\mathcal{F} = (M, \{A^q\})$ consisting of a leading DFA $M = (\Sigma, Q, \bar{q}, \mathrm{T}, \emptyset)$ and of a progress DFA $A^q = (\Sigma, Q^q, \bar{q}^q, \mathrm{T}^q, F^q)$ for each $q \in Q$.*

In the remainder of the paper we may just write $M = (\Sigma, Q, \bar{q}, \mathrm{T})$ for a leading DFA. We say that a decomposition (u, v) is accepted by an FDFA $\mathcal{F}$ if $M(u) = M(uv)$ and $A^q(v) \in F^q$ where $q = M(u)$. An ultimately periodic word $\alpha \in \Sigma^\omega$ is accepted by an FDFA $\mathcal{F}$ if there exists a decomposition (u, v) of α that is accepted by $\mathcal{F}$. Then we define $\mathrm{UP}(\mathcal{F}) = \{\, \alpha \in \Sigma^\omega \mid \alpha \text{ is accepted by } \mathcal{F} \,\}$. As an example of FDFAs, consider the FDFA $\mathcal{F}$ shown in Figure 2: the leading DFA M has only one state, ε, and the corresponding progress DFA for state ε is A^ε. The word ab^ω is accepted by $\mathcal{F}$ since there exists the decomposition (a, b) of ab^ω being accepted by $\mathcal{F}$. It is easy to see that $\mathrm{UP}(\mathcal{F}) = \Sigma^* \cdot b^\omega$, which is also recognised by the NBA $\mathcal{B}$ depicted in Figure 2.

In [8], Angluin and Fisman propose to use three canonical FDFAs to recognise ω-regular languages, namely periodic FDFAs, syntactic FDFAs, and recurrent FDFAs. In this paper, we only use syntactic FDFAs since they can be exponentially smaller than their periodic counterpart [8] and have proved to be well suited for converting to Büchi automata [35]. The right congruence $\sim_L$ of a given ω-regular language L is defined such that $x \sim_L y$ if for each $w \in \Sigma^\omega$, it holds that $xw \in L$ if and only if $yw \in L$.

Definition 4 (Syntactic FDFA [8]). *Given an ω-regular language L, the syntactic FDFA $\mathcal{F} = (M, \{A^u\})$ for L is defined as follows. The leading DFA M is the tuple $M = (\Sigma, \Sigma^*/_{\sim_L}, [\varepsilon]_{\sim_L}, \mathrm{T}, \emptyset)$, where $\mathrm{T}([u]_{\sim_L}, a) = [ua]_{\sim_L}$ for all $u \in \Sigma^*$ and $a \in \Sigma$.*

The right congruence $\approx^u_S$ for a progress DFA A^u of the syntactic FDFA is defined as follows.

$$x \approx^u_S y \text{ iff } ux \sim_L uy \wedge \forall v \in \Sigma^*. uxv \sim_L u \implies (u(xv)^\omega \in L \iff u(yv)^\omega \in L).$$

The progress DFA A^u is the tuple $(\Sigma, \Sigma^*/_{\approx^u_S}, [\varepsilon]_{\approx^u_S}, \mathrm{T}_S, F_S)$, where, for each $v \in \Sigma^*$ and $a \in \Sigma$, $\mathrm{T}_S([v]_{\approx^u_S}, a) = [va]_{\approx^u_S}$. The set of accepting states F_S is the set of equivalence classes $[v]_{\approx^u_S}$, for which $uv \sim_L u$ and $uv^\omega \in L$ hold.

Given an ω-regular language L, the corresponding syntactic FDFA for L has finite states [8].

Lemma 1 (cf. [8]). *Let $\mathcal{F} = (M, \{A^q\})$ be a syntactic FDFA recognising the ω-regular language L. Then we have $UP(\mathcal{F}) = UP(L)$ and if $xy^\omega \in L$, then every decomposition (u, v) of xy^ω with $M(u) = M(uv)$ is accepted by $\mathcal{F}$.*

An example of syntactic FDFAs is the FDFA $\mathcal{F}$ shown in Figure 2. This FDFA $\mathcal{F}$ recognises the ω-regular language $\Sigma^* \cdot b^\omega$. Since ab^ω is accepted by $\mathcal{F}$, we have that every decomposition of ab^ω is accepted by $\mathcal{F}$.

Definition 5 (Complement of FDFA [7]). *Given an FDFA $\mathcal{F} = (M, \{A^q\})$, the* complement $\mathcal{F}^\mathsf{c}$ *of $\mathcal{F}$ is the FDFA $\mathcal{F}^\mathsf{c} = (M, \{A^{q\mathsf{c}}\})$.*

In contrast to [7], we consider general FDFAs instead of only canonical FDFAs in this paper. As a consequence, though we call $\mathcal{F}^\mathsf{c}$ the complement of $\mathcal{F}$, actually it is possible to have $\mathrm{UP}(\mathcal{F}) \cap \mathrm{UP}(\mathcal{F}^\mathsf{c}) \neq \emptyset$. This complicates the use of $\mathcal{F}^\mathsf{c}$. More details will be given in Section 3.

Transforming FDFAs to Büchi Automata. According to [35, Section 6], an FDFA $\mathcal{F}$ does not necessarily recognise an ω-regular language. Thus one cannot construct an NBA $\mathcal{B}$ for an arbitrary FDFA $\mathcal{F}$ such that $\mathrm{UP}(\mathcal{F}) = \mathrm{UP}(\mathcal{L}(\mathcal{B}))$. To overcome this obstacle, the authors of [35] propose two methods to approximate $\mathrm{UP}(\mathcal{F})$ by means of two Büchi automata $\underline{B}(\mathcal{F})$ and $\overline{B}(\mathcal{F})$ that accept an under- and an over-approximation, respectively, of $\mathrm{UP}(\mathcal{F})$. We use the under-approximation method, because this ensures that $\mathrm{UP}(\mathcal{L}(\underline{B}(\mathcal{F}))) = \mathrm{UP}(\mathcal{F})$ holds whenever $\mathcal{F}$ is a canonical FDFA (cf. [35, Lemma 3]). No such property has been established for the over-approximation method.

We now present the idea underlying the construction of the under-approximation $\underline{B}(\mathcal{F})$ proposed in [35], to which we refer for details. Recall that A^s_f denotes the DFA A where s is the initial state and f the only accepting state; recall that an FDFA $\mathcal{F} = (M, \{A^q\})$ consists of a leading DFA $M = (\Sigma, Q, \bar{q}, \mathrm{T}, \emptyset)$ and of a progress DFA $A^q = (\Sigma, Q^q, \bar{q}^q, \mathrm{T}^q, F^q)$ for each $q \in Q$; recall also that $\mathrm{UP}(\mathcal{F}) = \{\, \alpha \in \Sigma^\omega \mid \alpha \text{ is accepted by } \mathcal{F} \,\}$, where α is accepted if there exists a decomposition (u, v) of α, such that $uv^\omega = \alpha$, $M(u) = M(uv)$, and $A^q(v) \in F^q$,

where $q = M(u)$. This implies that every word α in UP($\mathcal{F}$) can be decomposed into two parts u and v, such that u is consumed by a run of M and v by a run of A^q. Note that, if we consider $M_q^{\bar{q}}$, then we have that uv^ω is accepted by $\mathcal{F}$ if $M_q^{\bar{q}}(u) = M_q^{\bar{q}}(uv)$, $u \in \mathcal{L}(M_q^{\bar{q}})$, and $A^q(v) \in F^q$, where $q = M(u)$. This means that we can write UP($\mathcal{F}$) as $\mathrm{UP}(\mathcal{F}) = \bigcup_{q \in Q, f \in F^q} \mathcal{L}(M_q^{\bar{q}}) \cdot N_{(q,f)}$ where $N_{(q,f)} = \{ v^\omega \in \Sigma^\omega \mid v \in \Sigma^+ \wedge q = M_q^q(v) \wedge v \in \mathcal{L}((A^q)_f^{\bar{q}^q}) \}$ is the set of all infinite repetitions of the finite words v accepted by A_f^q.

In order to under-approximate UP($\mathcal{F}$), it is enough to match exactly $\mathcal{L}(M_q^{\bar{q}})$ and to under-approximate $N_{(q,f)}$. The former is trivial, since we already have $M_q^{\bar{q}}$; for the latter, consider the DFA $\underline{P}_{(q,f)} = M_q^q \times (A^q)_f^{\bar{q}^q} \times (A^q)_f^f$, where $\times$ stands for the standard intersection product between DFAs: the DFA $M_q^q \times (A^q)_f^{\bar{q}^q}$ ensures that for any $v \in \mathcal{L}(M_q^q \times (A^q)_f^{\bar{q}^q})$ and $u \in \mathcal{L}(M_q^{\bar{q}})$, we have $q = M(u) = M(uv)$ while $(A^q)_f^f$ guarantees that $v, v^2 \in \mathcal{L}((A^q)_f^{\bar{q}^q})$. Then, by the construction in [35, Definition 4], it is possible to construct an NBA $\underline{B}(\mathcal{F})$ such that $\mathcal{L}(\underline{B}(\mathcal{F})) = \bigcup_{q \in Q, f \in F^q} \mathcal{L}(M_q^{\bar{q}}) \cdot \underline{N}_{(q,f)}$ where $\underline{N}_{(q,f)} = \mathcal{L}(\underline{P}_{(q,f)})^\omega$. $\underline{B}(\mathcal{F})$ under-approximates the language of $\mathcal{F}$:

Lemma 2 ([35, Lemma 3]). *For every FDFA $\mathcal{F}$, UP($\mathcal{L}(\underline{B}(\mathcal{F}))$) $\subseteq$ UP($\mathcal{F}$) holds. If $\mathcal{F}$ is canonical, then UP($\mathcal{L}(\underline{B}(\mathcal{F}))$) $=$ UP($\mathcal{F}$) holds.*

3 Learning to Complement Büchi Automata

In this section we present the details of our learning framework, depicted in Figure 1, to learn the complement language $L = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$ of a given NBA $\mathcal{B}$. We first outline the general framework. We then continue with the technical part, where we first show that the counterexamples are correct in Section 3.2, and then establish termination and correctness in Section 3.3, before we finally discuss an optimisation in Section 3.4.

3.1 The Learning Framework

We begin with an introduction of the learning framework for L, depicted in Figure 1. The framework consists of two components, namely the learner and the teacher for complementing Büchi automaton.

The learner is a standard FDFA learner (see, e.g. [8,35]). He tries to learn an FDFA that recognises an ω-regular language L by means of two types of queries: membership queries of the form MQ(u, v) that provide him with information about whether the word uv^ω has to be included in L; and equivalence queries of the form EQ($\mathcal{F}$), aimed to find differences between the current conjecture $\mathcal{F}$ and the language he shall learn. The learner is oblivious of the fact that the NBA $\underline{B}(\mathcal{F})$ is sought after, not $\mathcal{F}$ itself.

The teacher provides answers to these queries based on a definition of the complement of L by an NBA $\mathcal{B}$. Answering a membership query MQ(u, v) is easy: it reduces to checking whether $uv^\omega \in L$, i.e. whether $uv^\omega \notin \mathcal{L}(\mathcal{B})$.

The innovation is in the way the equivalence queries $\text{EQ}(\mathcal{F})$ are answered. For checking equivalence, the teacher works with two NBAs $\underline{B}(\mathcal{F})$ and $\underline{B}(\mathcal{F}^c)$ that underestimate the ω-languages recognised by $\mathcal{F}$ and $\mathcal{F}^c$, respectively. She reports equivalence to the learner, when she is satisfied that $\mathcal{L}(\underline{B}(\mathcal{F})) = L$ holds. For algorithmic reasons, this is the case when $\mathcal{L}(\underline{B}(\mathcal{F}^c)) = \mathcal{L}(\mathcal{B})$ holds, too.

In her first step in answering an equivalence query $\text{EQ}(\mathcal{F})$, she constructs the NBA $\underline{B}(\mathcal{F})$ from the conjecture $\mathcal{F}$ and then checks whether $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$ holds. If this is not the case, then a witness $xy^\omega \in \mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B})$ is constructed. Since $\text{UP}(\mathcal{L}(\underline{B}(\mathcal{F}))) \subseteq \text{UP}(\mathcal{F})$ is established in Lemma 2, this implies $xy^\omega \in \text{UP}(\mathcal{F}) \cap \text{UP}(\mathcal{L}(\mathcal{B}))$.

She then analyses the witness xy^ω to get a decomposition (u, v) of xy^ω that is accepted by $\mathcal{F}$. She then returns (u, v) to the learner as a counterexample (that matches Definition 6), for him to remove (u, v) from the current FDFA $\mathcal{F}$, since $uv^\omega \in \mathcal{L}(\mathcal{B})$.

When the first check $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$ has been passed successfully, the teacher constructs $\underline{B}(\mathcal{F}^c)$ and checks whether $\mathcal{L}(\underline{B}(\mathcal{F}^c)) \subseteq \mathcal{L}(\mathcal{B})$ holds. This language inclusion test is delegated to the off-the-shelf tool RABIT [3, 4, 52]. Note that RABIT does not complement either of the two input languages. If language inclusion holds, we exploit $\mathcal{L}(\underline{B}(\mathcal{F})) \cup \mathcal{L}(\underline{B}(\mathcal{F}^c)) = \Sigma^\omega$ (a property we establish in Proposition 2) to infer $\mathcal{L}(\underline{B}(\mathcal{F})) \cup \mathcal{L}(\mathcal{B}) = \Sigma^\omega$. Since we know that $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$ holds from the first check, this implies that $\underline{B}(\mathcal{F})$ complements $\mathcal{B}$.

If the second check fails, the teacher gets a witness $xy^\omega \in \mathcal{L}(\underline{B}(\mathcal{F}^c)) \setminus \mathcal{L}(\mathcal{B})$, such that $\text{UP}(\mathcal{L}(\underline{B}(\mathcal{F}^c))) \subseteq \text{UP}(\mathcal{F}^c)$ (Lemma 2) implies $xy^\omega \in \text{UP}(\mathcal{F}^c) \setminus \text{UP}(\mathcal{L}(\mathcal{B}))$. She then analyses the witness xy^ω to derive a decomposition (u, v) of xy^ω that is accepted by $\mathcal{F}^c$. She then returns (u, v) to the learner as a counterexample (that matches Definition 6), for him to add (u, v) to the current FDFA $\mathcal{F}$, since $uv^\omega \notin \mathcal{L}(\mathcal{B})$ and (u, v) is not accepted by $\mathcal{F}$.

3.2 Correctness of the Counterexample Analysis

One important task of the teacher in the learning framework depicted in Figure 1 is the construction of the appropriate counterexample (u, v) in case the equivalence query $\text{EQ}(\mathcal{F})$ has to be answered negatively. Note that this is the only step in our learning loop that depends on the representation of the complement language by $\mathcal{B}$—a much looser connection than for the off-the-shelf complementation algorithms implemented in GOAL [54] and SPOT [24]. The counterexample we receive is an ultimately periodic word xy^ω. We cannot, however, simply return (x, y) but we have to infer an appropriate counterexample (u, v) such that $uv^\omega = xy^\omega$. For this, we first recall the notion of counterexamples for FDFA learners.

Definition 6 (Counterexample for the FDFA learner [35]). *Given a conjectured FDFA $\mathcal{F} = (M, \{A^q\})$ and the target language L, we say that a counterexample (u, v) is*

- *positive if $M(u) = M(uv)$, $uv^\omega \in UP(L)$, and (u, v) is not accepted by $\mathcal{F}$,*

– *negative if* $M(u) = M(uv)$, $uv^\omega \notin UP(L)$, *and* (u, v) *is accepted by* $\mathcal{F}$.

Note that, when a pair (u, v) is accepted by $\mathcal{F}$, then $M(u) = M(uv)$ holds. The FDFA learner underlying the Büchi automaton complementation learner can use the counterexample for the FDFA learner to refine the conjecture $\mathcal{F}$ for the target language L. Intuitively, if a counterexample (u, v) is positive, then $\mathcal{F}$ should accept it, while $\mathcal{F}$ should reject it when it is negative. Our goal is to infer a valid decomposition (u, v) from xy^ω, which matches the cases in Definition 6, to be able to refine $\mathcal{F}$. Proposition 1 guarantees that, if there exists xy^ω violating the checks performed in our learning framework, then we can always construct a decomposition (u, v) from xy^ω—that satisfies $uv^\omega = xy^\omega$—to refine $\mathcal{F}$.

Proposition 1. *Given an NBA* $\mathcal{B}$ *with alphabet* Σ, *let* $L = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$ *be its complement language and target* ω*-regular language. Suppose* $\mathcal{F}$ *is the current FDFA conjecture. Whenever the teacher returns* (u, v) *as answer to an equivalence query* $\mathrm{EQ}(\mathcal{F})$, *then* (u, v) *is either a positive or negative counterexample.*

3.3 Termination and Correctness of the Learning Algorithm

Based on Proposition 1, the learner can refine the current FDFA $\mathcal{F}$ with the returned counterexample (u, v) from the teacher. Since the learner is the same as the FDFA learner proposed in [8, 35], in the worst case, we have to get the canonical FDFA that recognises L in order to complete the learning task. Moreover, the number of membership queries and equivalence queries are polynomial in the size of the canonical periodic FDFA [8, 35].

In order to establish the correctness of our learning algorithm, we first introduce a result that, while being used for proving the correctness of the algorithm, is of interest in its own right: we establish in Proposition 2 that, for a (not necessarily canonical) FDFA $\mathcal{F}$, the NBAs $\underline{B}(\mathcal{F})$ and $\underline{B}(\mathcal{F}^c)$ that underapproximate the languages of $\mathcal{F}$ and its complement $\mathcal{F}^c$, respectively, cover the whole Σ^ω. This generalises a simpler result for canonical FDFAs from [7].

Proposition 2. *Given an FDFA* $\mathcal{F}$ *with alphabet* Σ, *it is* $\mathcal{L}(\underline{B}(\mathcal{F})) \cup \mathcal{L}(\underline{B}(\mathcal{F}^c)) = \Sigma^\omega$.

Proof. First one can show that for each pair of ω-regular languages L_1 and L_2, we have that $\mathrm{UP}(L_1 \cup L_2) = \mathrm{UP}(L_1) \cup \mathrm{UP}(L_2)$. By Theorem 1, it suffices to prove that $\mathrm{UP}(\mathcal{L}(\underline{B}(\mathcal{F}))) \cup \mathrm{UP}(\mathcal{L}(\underline{B}(\mathcal{F}^c))) = \mathrm{UP}(\Sigma^\omega) = \{ uv^\omega \in \Sigma^\omega \mid u \in \Sigma^*, v \in \Sigma^+ \}$ holds in order to show that $\mathcal{L}(\underline{B}(\mathcal{F})) \cup \mathcal{L}(\underline{B}(\mathcal{F}^c)) = \Sigma^\omega$ holds. That is, we need to show that, for all finite words $u \in \Sigma^*$ and $v \in \Sigma^+$, $uv^\omega \in \mathrm{UP}(\mathcal{L}(\underline{B}(\mathcal{F})))$ or $uv^\omega \in \mathrm{UP}(\mathcal{L}(\underline{B}(\mathcal{F}^c)))$.

Given an FDFA $\mathcal{F} = (M, \{A^q\})$, for any $u \in \Sigma^*$ and $v \in \Sigma^+$, by [8] we can always find a *normalised* decomposition (x, y) of uv^ω such that $q = M(x) = M(xy)$ and $xy^\omega = uv^\omega$ since M is a complete DFA with a finite set of states. Then, one can show that there exists some $j \geq 1$ such that y^j is either accepted by A^q or $A^{q\mathsf{c}}$. Therefore, we can conclude that (x, y^j) is either accepted by $\mathcal{F}$ or $\mathcal{F}^c$. Consequently, we get that $xy^\omega = x(y^j)^\omega \in \mathrm{UP}(\mathcal{L}(\underline{B}(\mathcal{F})))$ or that $xy^\omega = x(y^j)^\omega \in \mathrm{UP}(\mathcal{L}(\underline{B}(\mathcal{F}^c)))$, as required. □

The following theorem guarantees the main result about the termination and correctness of the proposed framework. That is, the learning algorithm always returns an NBA that accepts the complement language of the given $\mathcal{B}$.

Theorem 2. *Given an NBA $\mathcal{B}$ with alphabet Σ, the learning algorithm depicted in Figure 1 terminates and returns an NBA $\underline{B}(\mathcal{F})$ such that $\mathcal{L}(\underline{B}(\mathcal{F})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.*

Note that the algorithm can terminate before we have learned the canonical FDFA that represents $\Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$: on termination $\mathcal{L}(\underline{B}(\mathcal{F})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$ is guaranteed since the conjecture $\mathcal{F}$ satisfies $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$ and $\mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) \subseteq \mathcal{L}(\mathcal{B})$. When a conjectured $\mathcal{F}$ does not satisfy $\mathcal{L}(\underline{B}(\mathcal{F})) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$, then it is easy to conclude, together with $\mathcal{L}(\underline{B}(\mathcal{F})) \cup \mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) = \Sigma^\omega$ by Proposition 2, that $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$ or $\mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) \not\subseteq \mathcal{L}(\mathcal{B})$ holds.

Corollary 1. *The learning algorithm terminates with $\mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) = \mathcal{L}(\mathcal{B}) = \Sigma^\omega \setminus \mathcal{L}(\underline{B}(\mathcal{F}))$.*

From Corollary 1, we can get a Büchi automaton $\underline{B}(\mathcal{F}^{\mathsf{c}})$ accepting the same language of $\mathcal{B}$ as a for-free by-product of the complementing algorithm. This means that we have also provided an alternative oracle that can be used to learn the language of $\mathcal{B}$, which can be another method to reduce the size of $\mathcal{B}$. Therefore, our learning based complementation algorithm has proven beneficial not to learn an FDFA that represents the complement language of a Büchi automaton (or the language itself, as complementing FDFAs is cheap), but to use it as an intermediate construction in the learning cycle.

3.4 An Improved Algorithm

Once the learning algorithm terminates we have that $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$ and $\mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) \subseteq \mathcal{L}(\mathcal{B})$. It trivially follows that $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) = \emptyset$ holds.

Corollary 2. *The learning algorithm terminates with $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) = \emptyset$.*

Therefore, $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) = \emptyset$ is a necessary condition for the termination of the learning framework. Since the most expensive step is checking language inclusion between $\mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}}))$ and $\mathcal{L}(\mathcal{B})$, we should avoid this check whenever possible. To do so, we can simply check whether $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}})) = \emptyset$ holds right before checking the language inclusion.

If there exists some $xy^\omega \in \mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\underline{B}(\mathcal{F}^{\mathsf{c}}))$, then we have in particular that some decomposition (u, v) of xy^ω is accepted by $\mathcal{F}^{\mathsf{c}}$, as well as $xy^\omega \in \mathcal{L}(\underline{B}(\mathcal{F}))$. The latter implies with $\mathcal{L}(\underline{B}(\mathcal{F})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$ (recall that this is checked first) that $xy^\omega \in L$ (since $\mathcal{L}(\underline{B}(\mathcal{F})) \subseteq L$ was shown). We can therefore return the decomposition (u, v) as a positive counterexample for the FDFA learner to refine $\mathcal{F}$. Otherwise, we just proceed to check the language inclusion.

This optimisation preserves the correctness of the algorithm, and we apply it by default.

Table 1. Comparison between GOAL, SPOT, and Buechic on complementing Büchi Store. The average number of letters in each alphabet is about 9.

Block	Experiments (States, Transitions)		GOAL				Buechic	SPOT
			Ramsey	Determinisation	Rank	Slice		
1	287 NBAs (928, 2071)	$\|Q\|$	21610	3919	21769	4537	2428	**1629**
		$\|T\|$	964105	87033	179983	125155	35392	**13623**
		t_c	992	300	203	204	105	**6**
2	5 NBAs (55, 304)	$\|Q\|$	–to–	926	38172	1541	**165**	495
		$\|T\|$		21845	384378	50689	5768	**4263**
		t_c		28	42	12	474	**<1**
3	2 NBAs (20, 80)	$\|Q\|$	–to–	–to–	27372	11734	**96**	2210
		$\|T\|$			622071	1391424	**6260**	102180
		t_c			56	152	7	**1**

4 Experimental Evaluation

We have implemented a prototype, Buechic, of our learning approach based on the ROLL learning library [35]. We use RABIT [3,4,52] to perform the inclusion check $\mathcal{L}(\underline{B}(\mathcal{F}^c)) \subseteq \mathcal{L}(\mathcal{B})$ that occurs in the evaluation of the equivalence query EQ($\mathcal{F}$) (cf. Figure 1). The machine we used for the experiments is a 3.6 GHz Intel i7-4790 with 16 GB of RAM, of which 8 GB were assigned to the tool. The timeout has been set to 300 seconds in this section. In the experiments, we compare our Büchi complementation algorithm with two tools. The first tool is GOAL (the latest version 2015-10-18) [54], which is a mature and well-known tool for manipulating Büchi automata. We consider four different complementing algorithms implemented in GOAL, see [54] for more details.

We have used SPOT (the stable version 2.3.5) [24] as a second point of comparison. SPOT is the state-of-the-art platform for manipulating ω-automata, including Büchi automata. Recall that SPOT does not provide a complementation function for generic Büchi automata directly, thus we first use SPOT to get a deterministic automaton from the given Büchi automaton, then complement the resulting deterministic automaton (for parity automata this simply means adding 1 to all priorities), and finally transform the resulting complement automaton to an equivalent Büchi automaton. (This follows one of the classic approaches for complementing Büchi automata.)

The automata we used in this section for the experiments are taken from the benchmark sets provided by Büchi Store [55] and the Büchi automata generated by SPOT from the formulas in [50]. The former contains 295 NBAs with 1 to 17 states and with 0 to 123 transitions; the latter comprises 90 NBAs with 1 to 165 states and with 0 to 493 transitions. We then considered 300 randomly generated Büchi automata generated by SPOT. All automata are represented in the Hanoi Omega-Automata (HOA) format [10].

4.1 Complementation for Büchi Store

Büchi Store provides 295 nondeterministic Büchi automata; however, since one of such automata has only one state without transitions and GOAL fails in recog-

nising it as a Büchi automaton, we decided to exclude it from the experiments and consider only the remaining 294 cases. In practice, such an automaton accepts the empty language, so its complement accepts the whole Σ^ω. Our tool learns a complement automaton with 3 states and 12 transitions in just 0.16 seconds, so it mildly contributes to demonstrate the efficiency of Buechic. SPOT can also output a complement automaton with 1 state and 1 transition in just 0.02 seconds, which is the smallest Büchi automaton recognising Σ^ω.

The experiments shown in Table 1 are organised by blocks of rows; each block reports the experiments it represents together with the total number of states and transitions of the considered input NBAs and comprises three rows, marked with $|Q|$, $|\mathrm{T}|$, t_c, reporting the overall number of states and transitions, and the total time in seconds, respectively, spent by the different tools for computing the complement automata. For each row, we mark in bold the minimum value among all entries.

By inspecting the entries in Table 1 we can see that our learning based complementation method always outperforms the complementation methods offered by GOAL when we consider the number of states and transitions. If we compare Buechic with SPOT, we can find that for 287 out of the 294 tasks, SPOT produces smaller complement automata than other competitors. Moreover, SPOT is generally faster than the other competitors on all tasks. The results are not surprising since SPOT has implemented a lot of optimisations to reduce the size of the automata and it makes use of very efficient data structure called OBDDs. We note that for Block 2 and Block 3, our complementation method produces much smaller automata than the other tools. We explain later why this happens.

Block 1 reports the results relative to 287 NBAs which can be solved by all algorithms. For those automata, the complement NBAs learned by Buechic have much fewer states and transitions than the automata constructed by the algorithms from GOAL. Moreover, our learning algorithm spent less time than the four complementation algorithms from GOAL. Since on average only 7 equivalence queries are needed for the learning procedure for each NBA and the size of the corresponding FDFA is small, our learning based complementation algorithms perform well for those cases. Nevertheless, SPOT is faster than our learning algorithms and even produces smaller automata. This is because that on average there are only 3.2 states in each Büchi automaton and the optimisations in SPOT work quite well in reducing the size of their deterministic automata as well as their complement Büchi automata.

Block 2 refers to five NBAs on which only the Ramsey-based complementation approach fails. The NBAs in this block induce quite large complement automata, as we can see from the other GOAL solvers, thus quite some work is required for constructing them; this means that a failure can be expected also because the approach is rather slow compared with the other GOAL approaches. This is justified by the fact that, as mentioned in [11], the Ramsey-based complementation is the first complementation method proposed by Büchi [15] and was later improved in [51]. Our approach is much slower than GOAL and SPOT since on average 56 equivalence queries for the learning algorithm are posed

before obtaining the appropriate conjecture $\mathcal{F}$. However, the complement automata we learned have much fewer states than all approaches implemented in GOAL—and even SPOT—since the corresponding FDFAs we learned are small. It is worth mentioning that the reduction optimisations in SPOT are less effective here since the constructed automata by SPOT are relatively large. In our experiments, more states in an automaton usually go along with more transitions. The constructed automata by SPOT have fewer transitions since SPOT merges all transitions which have the same source state and target state as one transition, which is different from GOAL and Buechic.

Block 3 contains two NBAs on which both Ramsey- and determinisation-based complementation fail. For one NBA, the determinisation method can complete in 430 seconds and returns a Büchi automaton with 243 states. Regarding the other NBA, the determinisation method cannot terminate in 600 seconds. The bottleneck in this case is the transformation of the NBA to a deterministic parity automaton. In this block, our learning algorithm learns much smaller automata than its competitors since the corresponding FDFAs are very small.

For the given automata of Block 2 and Block 3, which are larger than the automata in Block 1, our algorithm can learn much smaller complement automata than its competitors. This is particularly important when the complementation is used by a model checker to check a system against a property that has been provided as a Büchi automaton or as an ω-regular language, since it helps in limiting the state-explosion problem the model checking algorithms are subject to.

4.2 Complementation for Büchi Automata Generated from Formulas

In order to compare our algorithms with GOAL and SPOT on larger Büchi automata than those in Büchi Store, we consider the Büchi automata generated by SPOT from the formulas in paper [50]. Table 2 gives the complementation results for the Büchi automata of 18 formulas that are explicitly given in [50]. From Table 2, we can conclude that our algorithm can learn much smaller automata than GOAL and SPOT on the large Büchi automata except for the formula pattern f(0, k) where $k \in \{0, 2, 4\}$.

We have also considered 72 further Büchi automata generated from 72 formulas from [50]. In summary, Ramsey-based, Determinisation-based, Rank-based and Slice-based GOAL approaches solve 49, 58, 61, and 62 complementation tasks, respectively, within the time limit, while SPOT solves 66 tasks and Buechic solves 65 tasks. The results are similar as those in Table 2; we thus only discuss the comparison between SPOT and Buechic as best performing tools. Note that there are 64 tasks solved by both SPOT and Buechic and those tasks solved only by SPOT and Buechic separately are disjoint, which implies that our algorithm complements existing complementation approaches very well. Due to the large number of cases, in order to present the experimental results in a more intuitive and compact way for all generated automata, we provide here the scatter plots of Buechic and SPOT in Figure 3 for the 64 commonly solved tasks.

Table 2. Comparison between GOAL, SPOT, and Buechic on complementing generated Büchi automata. The average number of letters in each alphabet is about 29.

Block	Experiments (States, Transitions)		GOAL				Buechic	SPOT
			Ramsey	Determinisation	Rank	Slice		
1	gf (j=1) (4, 8)	$\|Q\|$	17	26	34	30	**4**	12
		$\|T\|$	75	95	167	156	**16**	40
		t_c	**<1**	**<1**	**<1**	**<1**	**<1**	**<1**
2	gf (j=2) (6,14)	$\|Q\|$	275	405	808	467	**12**	82
		$\|T\|$	4609	5881	11408	9440	**158**	736
		t_c	10	6	2	3	**<1**	**<1**
3	gf (j=3) (8,21)	$\|Q\|$			25248	15691	**34**	1547
		$\|T\|$	–to–	–to–	1097400	1270016	**1512**	60973
		t_c			99	175	**2**	**2**
4	gf (j=4) (10,29)	$\|Q\|$					**95**	
		$\|T\|$	–to–	–to–	–to–	–to–	**14922**	–to–
		t_c					**45**	
5	gffg1 (j=2) (9,22)	$\|Q\|$	559	1497	5773	1333	**22**	242
		$\|T\|$	9027	17386	44277	25760	**210**	2613
		t_c	48	19	6	6	**<1**	**<1**
6	gffg2 (j=3) (21,59)	$\|Q\|$					**118**	20558
		$\|T\|$	–to–	–to–	–to–	–to–	**2662**	806312
		t_c					**6**	9
7	phi1 (4,10)	$\|Q\|$	17	5	5	5	**4**	5
		$\|T\|$	77	15	22	22	16	**11**
		t_c	**<1**	**<1**	**<1**	**<1**	**<1**	**<1**
8	phi2 (4,10)	$\|Q\|$	31	21	32	19	**3**	8
		$\|T\|$	149	70	126	84	**9**	22
		t_c	**<1**	**<1**	**<1**	**<1**	**<1**	**<1**
9	phi3 (6,14)	$\|Q\|$	39	6	9	8	**4**	6
		$\|T\|$	165	19	38	36	16	**13**
		t_c	1	**<1**	**<1**	**<1**	**<1**	**<1**
10	f(0, 0) (5,9)	$\|Q\|$	29	**5**	7	7	8	**5**
		$\|T\|$	135	20	32	32	36	**9**
		t_c	**<1**	**<1**	**<1**	**<1**	**<1**	**<1**
11	f(0, 2) (9,13)	$\|Q\|$	214	13	141	56	10	**7**
		$\|T\|$	1164	51	538	268	44	**11**
		t_c	37	**<1**	**<1**	**<1**	**<1**	**<1**
12	f(0, 4) (13,17)	$\|Q\|$		15	234	101	12	**9**
		$\|T\|$	–to–	59	854	456	52	**13**
		t_c		**<1**	**<1**	**<1**	**<1**	**<1**
13	f(1, 0) (12,35)	$\|Q\|$		105	8121	581	**10**	100
		$\|T\|$	–to–	534	36685	5096	**69**	563
		t_c		1	5	2	2	**<1**
14	f(1, 2) (31,88)	$\|Q\|$			83050	8413	**16**	175
		$\|T\|$	–to–	–to–	367699	82832	**109**	1034
		t_c			86	25	**<1**	**<1**
15	f(1, 4) (107, 306)	$\|Q\|$					**18**	2723
		$\|T\|$	–to–	–to–	–to–	–to–	**125**	20442
		t_c					**2**	3
16	f(2, 0) (18,56)	$\|Q\|$				31281	**10**	840
		$\|T\|$	–to–	–to–	–to–	529216	**133**	6460
		t_c				103	26	**<1**
17	f(2, 2) (47,141)	$\|Q\|$					**9**	1161
		$\|T\|$	–to–	–to–	–to–	–to–	**144**	10630
		t_c					2	**<1**
18	f(2, 4) (165, 493)	$\|Q\|$						
		$\|T\|$	–to–	–to–	–to–	–to–	–to–	–to–
		t_c						

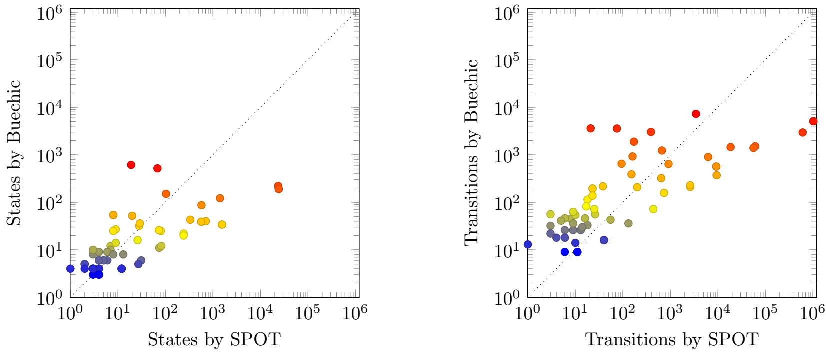


Fig. 3. Comparison between the number of states and transitions of automata generated by SPOT and Buechic on 72 automata corresponding to formulas from [50]. The average number of letters in each alphabet is about 301.

In Figure 3, the coordinate values of the y axis and x axis are the corresponding number of states (resp. transitions) in the complement automata of Buechic and SPOT. All points below the dotted diagonal indicate that the complement automata learned by our algorithm have smaller values than the complement automata constructed by SPOT, which is the case for almost all large examples. We recall that SPOT merges transitions that share the same source state and target state as one transition, so in the right scatter plot of Figure 3, many points are above the diagonal line. Nevertheless, we can learn from the plots that only SPOT produces those automata with more than 10^4 transitions, which indicates that the reduction optimisations of SPOT do not work well on large automata and our algorithm performs much better on large automata. Figure 4 is similar to Figure 3 but it refers to 300 randomly generated Büchi automata with size ranging between 1 and 69 states and between 0 and 263 transitions. The behaviour of SPOT on these automata is similar to the one shown in Figure 3.

In order to show how the growing trend of the number of states in the complement automata of the complementation algorithms behaves when we increase the size of the given Büchi automata in some cases, we take the generated Büchi automata of the formula pattern $\bigwedge_{i=1}^{k}(\mathsf{GF}a_i) \to \mathsf{GF}b$. The growing trend of the number of states in the complement automata for the approaches in GOAL, SPOT, and Buechic are plotted in Figure 5. The number of states in the complement automaton constructed by GOAL and SPOT is growing exponentially with respect to the parameter k, while the number of states in the complement automaton learned by our learning algorithm grows much more slowly than others. The experimental results show that the performance of our algorithm can be much more stable for some automata with their growth of the states. Thus an advantage of our learning approach is that it has potentially better performance on large automata compared to classic complementation techniques.

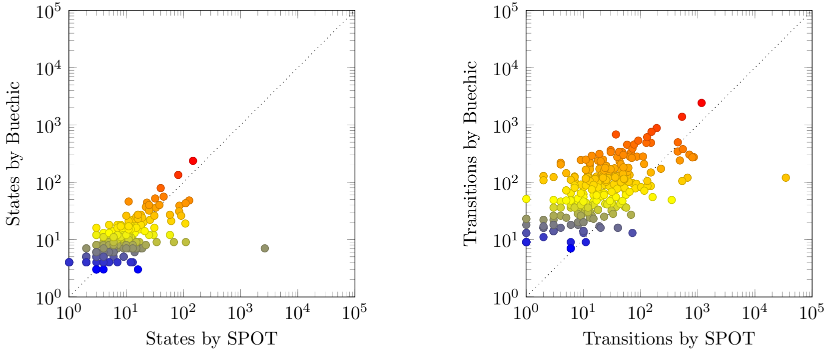


Fig. 4. Comparison between the number of states and transitions of automata constructed by SPOT and Buechic on 300 randomly generated automata. The average number of letters in each alphabet is about 7.

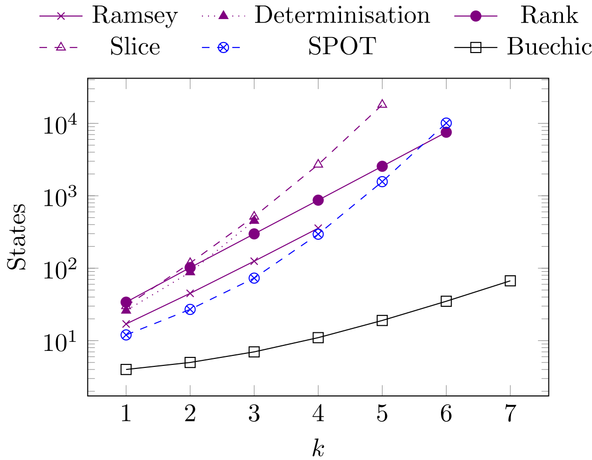


Fig. 5. States comparison of GOAL, SPOT, and Buechic on the formula pattern $\bigwedge_{i=1}^{k}(\mathsf{GF}a_i) \rightarrow \mathsf{GF}b$. The number of letters in the alphabet is 2^{k+1} for case k.

4.3 Further Experimental Results

We have conducted further experiments. We have considered *double* complementation on the automata from Büchi Store and generated automata by SPOT. We define double complementation as first using a complementation algorithm to complement the input NBA $\mathcal{B}$ to get the complement $\mathcal{B}^{\mathsf{c}}$ of $\mathcal{B}$; and then complementing $\mathcal{B}^{\mathsf{c}}$ using the same algorithm. It is actually an empty operation. From the experiments, in particular where the complement automata were large, we gained advantage over the competitor algorithms. As another set of

benchmarks, we have also considered the complementation of semi-deterministic automata (sometimes called limit-deterministic automata). We considered all 106 SDBAs from [11], and additionally compared them with the NCSB method from [11], which is implemented in GOAL. Note that this is a specialist method and we compete on its soil. This becomes quite clear when comparing with the other general complementation techniques. The experimental evaluation shows that we are competitive with the specialised method from [11] and the highly optimised tool SPOT. Finally, we considered a variation of our learning algorithm, that is, we experimented with checking completeness of $\mathcal{L}(\underline{B}(\mathcal{F})) \cup \mathcal{L}(\mathcal{B})$ instead of testing language inclusion $\mathcal{L}(\underline{B}(\mathcal{F}^c)) \subseteq \mathcal{L}(\mathcal{B})$, as proposed in Figure 1. The universality check for $\mathcal{L}(\underline{B}(\mathcal{F})) \cup \mathcal{L}(\mathcal{B})$ is only invoked after the disjointness test for $\mathcal{L}(\underline{B}(\mathcal{F}))$ and $\mathcal{L}(\mathcal{B})$ is passed. According to the experimental results, our handling with the equivalence queries in Figure 1 is more practical.

5 Conclusion

We have introduced a learning based approach for the complementation of Büchi automata. We expected that learning based approaches provide small complementations, that they are less perceptive of the initial representation of the ω-regular language to complement, and that they tend to be fast. In short: that they are practical.

Our experimental evaluation has confirmed our expectation that learning based complementation usually provides smaller complements. More surprisingly, the language inclusion checks in the loop are usually quite fast. As a result, the running time displayed by Buechic is competitive. We have also seen that, while we did gain a clear advantage over the basic techniques as implemented in GOAL, the comparison with SPOT shows that this advantage is not quite in the same league as the advantages one can obtain by high level optimisations implemented in SPOT. We expect that, after the pure technique has proven to be a very strong competitor, many improvements will follow. One improvement is to make the approach symbolic since learning algorithms usually become slow when dealing with large alphabets. This needs a symbolic learning algorithm for FDFAs, which is an interesting future work.

Acknowledgement. This work has been supported by the National Natural Science Foundation of China (Grants 61532019, 61472473, 61650410658), the CAS/SAFEA International Partnership Program for Creative Research Teams, the CAS Fellowship for International Young Scientists, the Sino-German CDZ project CAP (GZ 1023), the EPSRC grants EP/M027287/1 and EP/P020909/1.

References

1. F. Aarts, P. Fiterau-Brostean, H. Kuppens, and F. W. Vaandrager. Learning register automata with fresh value generation. In *ICTAC*, volume 9399 of *LNCS*, pages 165–183, 2015.

2. F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.
3. P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In *CAV*, volume 6174 of *LNCS*, pages 132–147, 2010.
4. P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Advanced Ramsey-based Büchi automata inclusion testing. In *CONCUR*, volume 6901 of *LNCS*, pages 187–202, 2011.
5. R. Alur, P. Černỳ, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
7. D. Angluin, U. Boker, and D. Fisman. Families of DFAs as acceptors of omega-regular languages. In *MFCS*, volume 58 of *LIPIcs*, pages 11:1–11:14, 2016.
8. D. Angluin and D. Fisman. Learning regular omega languages. *Theoretical Computer Science*, 650:57–72, 2016.
9. A. Arnold. A syntactic congruence for rational ω-languages. *Theoretical Computer Science*, 39:333–335, 1985.
10. T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Kretínský, D. Müller, D. Parker, and J. Strejcek. The Hanoi omega-automata format. In *CAV*, volume 9206 of *LNCS*, pages 479–486, 2015.
11. F. Blahoudek, M. Heizmann, S. Schewe, J. Strejček, and M.-H. Tsai. Complementing semi-deterministic Büchi automata. In *TACAS*, volume 9636 of *LNCS*, pages 770–787, 2016.
12. B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In *IJCAI*, pages 1004–1009, 2009.
13. M. Botincan and D. Babic. Sigma*: symbolic learning of input-output specifications. In *POPL*, pages 443–456, 2013.
14. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE TRANSACTIONS ON COMPUTERS*, 1986.
15. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Int. Congress on Logic, Methodology and Philosophy of Science*, pages 1–11, 1962.
16. H. Calbrix, M. Nivat, and A. Podelski. Ultimately periodic words of rational ω-languages. In *MFPS*, volume 802 of *LNCS*, pages 554–566, 1993.
17. S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *CAV*, volume 3576 of *LNCS*, pages 534–547, 2005.
18. S. Chaki and A. Gurfinkel. Automated assume-guarantee reasoning for omega-regular systems and specifications. *Innovations in Systems and Software Engineering*, 7:131–139, 2011.
19. S. Chaki and O. Strichman. Optimized L*-based assume-guarantee reasoning. In *TACAS*, volume 4424 of *LNCS*, pages 276–291, 2007.
20. M. Chapman, H. Chockler, P. Kesseli, D. Kroening, O. Strichman, and M. Tautschnig. Learning the language of error. In *ATVA*, volume 9364 of *LNCS*, pages 114–130, 2015.
21. Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning minimal separating DFAs for compositional verification. In *TACAS*, volume 5505 of *LNCS*, pages 31–45, 2009.

22. Y.-F. Chen, C. Hsieh, O. Lengál, T.-J. Lii, M.-H. Tsai, B.-Y. Wang, and F. Wang. PAC learning-based verification and model synthesis. In *ICSE*, pages 714–724, 2016.
23. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *LNCS*, pages 331–346, 2003.
24. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA*, volume 9938 of *LNCS*, pages 122–129. Springer, 2016.
25. A. Farzan, Y.-F. Chen, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *TACAS*, volume 4963 of *LNCS*, pages 2–17, 2008.
26. L. Feng, M. Kwiatkowska, and D. Parker. Compositional verification of probabilistic systems using learning. In *QEST*, pages 133–142, 2010.
27. L. Feng, M. Kwiatkowska, and D. Parker. Automated learning of probabilistic assumptions for compositional reasoning. In *FASE*, volume 6603 of *LNCS*, pages 2–17, 2011.
28. S. Fogarty, O. Kupferman, T. Wilke, and M. Y. Vardi. Unifying Büchi complementation constructions. *Logical Methods in Computer Science*, 9(1), 2013.
29. E. Friedgut, O. Kupferman, and M. Y. Vardi. Büchi complementation made tighter. *International Journal of Foundations of Computer Science*, 17(4):851–868, 2006.
30. S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. On complementing nondeterministic Büchi automata. In *CHARME*, volume 2860 of *LNCS*, pages 96–110, 2003.
31. F. He, X. Gao, B.-Y. Wang, and L. Zhang. Leveraging weighted automata in compositional reasoning about concurrent probabilistic systems. In *POPL*, pages 503–514, 2015.
32. M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, USA, 1994.
33. O. Kupferman and M. Y. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic*, 2(2):408–429, July 2001.
34. R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, 1994.
35. Y. Li, Y.-F. Chen, L. Zhang, and D. Liu. A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. In *TACAS*, volume 10205 of *LNCS*, pages 208–226, 2017.
36. O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1995.
37. O. Maler and L. Staiger. On syntactic congruences for omega-languages. In *STACS*, volume 665 of *LNCS*, pages 586–594, 1993.
38. R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.
39. M. Michel. Complementation is more difficult with automata on infinite words. Technical report, CNET, Paris (Manuscript), 1988.
40. D. E. Muller. Infinite sequences and finite machines. In *FOCS*, pages 3–16, 1963.
41. C. S. Păsăreanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.
42. J.-P. Pécuchet. On the complementation of Büchi automata. *Theoretical Computer Science*, 47(3):95–98, 1986.

43. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2001.
44. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *STOC*, pages 411–420, 1989.
45. S. Safra. On the complexity of omega-automata. In *FOCS*, pages 319–327, 1988.
46. W. J. Sakoda and M. Sipser. Non-determinism and the size of two-way automata. In *STOC*, pages 274–286, 1978.
47. S. Schewe. Büchi complementation made tight. In *STACS*, volume 3 of *LIPIcs*, pages 661–672, 2009.
48. S. Schewe and T. Varghese. Tight bounds for the determinisation and complementation of generalised Büchi automata. In *ATVA*, volume 7561 of *LNCS*, pages 42–56, 2012.
49. S. Schewe and T. Varghese. Tight bounds for complementing parity automata. In *MFCS*, volume 8634 of *LNCS*, pages 499–510, 2014.
50. S. Sickert, J. Esparza, S. Jaax, and J. Křetínský. Limit-deterministic Büchi automata for linear temporal logic. In *CAV*, volume 9780 of *LNCS*, pages 312–332, 2016.
51. A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(3):217–239, 1987.
52. R. tool. http://languageinclusion.org/doku.php?id=tools, 2016.
53. M. Tsai, S. Fogarty, M. Y. Vardi, and Y. Tsay. State of Büchi complementation. *Logical Methods in Computer Science*, 10(4), 2014.
54. M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. GOAL for games, omega-automata, and logics. In *CAV*, volume 8044 of *LNCS*, pages 883–889, 2013.
55. Y.-K. Tsay, M.-H. Tsai, J.-S. Chang, and Y.-W. Chang. Büchi store: An open repository of Büchi automata. In *TACAS*, volume 6605 of *LNCS*, pages 262–266, 2011.
56. M. Y. Vardi. The Büchi complementation saga. In *STACS*, volume 4393 of *LNCS*, pages 12–22, 2007.
57. Q. Yan. Lower bounds for complementation of ω-automata via the full automata technique. *Logical Methods in Computer Science*, 4(1:5), 2008.

P^5: Planner-less Proofs of Probabilistic Parameterized Protocols *

Lenore D. Zuck[1], Kenneth L. McMillan[2], and Jordan Torf[1]

[1] University of Illinois at Chicago, jtorf2@uic.edu, lenore@cs.uic.edu
[2] Microsoft Research, kenmcmil@microsoft.com

Abstract. Liveness of many probabilistic parameterized protocols are proven by first crafting a family of sequences of "good" random draws, thus, in effect "de-probabilizing" the system, and then proving the system just as one would for a non-probabilistic parameterized system. The family of "good" random draws (known in different names, such as "planner" and "strategy") is often an intricate piece of machinery, arising from the need to reason about a parameterized Markov Decision Process (MDP). In effect, it represents a parameterized strategy for an infinite game played between a probabilistic player and a non-deterministic adversary.

We present a novel approach to the problem that avoids the need to de-probabilize the system. First, we abstract the parameterized MDP to a finite MDP. The probabilistic choices of this abstraction are drawn not from an independent identically distributed random variable, but instead from a parameterized Markov chain. That is, the distribution of the random variable at any time is dependent on its history and also on the system's parameters. Then, we prove properties about infinite behaviors of the Markov chain and transfer these to the finite MDP. At this point, the proof can be completed by ordinary finite-state model checking. By using abstraction to separate parameterization from nondeterminism, we eliminate the parameterized game and avoid the need for a planner.

1 Introduction

Probabilistic elements were introduced into concurrent systems in the early 1980s to provide solutions (with high probability) to problems that do not have deterministic solutions. Among the pioneers of probabilistic protocols are [19, 29]. One of the most challenging problems in the study of probabilistic protocols has been their formal verification. While methodologies for proving safety (invariance) properties still hold for probabilistic protocols, formal verification of their liveness properties has been, and still remains, elusive. The main difficulty stems from the two types of nondeterminism that occur in such programs: Their asynchronous execution, that assumes a hostile (though somewhat fair) scheduler, and the nondeterminism associated with the probabilistic actions, that assumes an even-handed scheduler.

It had been realized that if one only wants to prove that a certain property is P-valid, that is, holds with probability 1 over all (fair) executions of a system, then for the case

* This work was partially funded by NSF award CCF-1563393

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 336–357, 2018.
https://doi.org/10.1007/978-3-319-73721-8_16

of finite-state system this can be attained in a manner that is independent of the precise probabilities. Decidability of P-validity of termination properties over finite-state systems had been first established in [12] using a methodology that is graph-theoretic in nature. The work in [26] extends the [12] method and presents deductive proof rules for proving P-validity for termination and progress properties of finite-state program. The work in [26] presents sound and complete methodology for establishing P-validity of general temporal properties over probabilistic systems, and [31, 26, 28] describe model checking procedure for the finite-state case.

The emerging interest in networked systems brought forth a surge of research in automatic verification of parameterized systems, that, having unbounded number of states, are not easily amenable to model checking techniques. In fact, verification of such systems is known to be undecidable [2] for important classes of systems. Much of the recent research has been devoted to identifying conditions that enable automatic verification of parameterized systems and abstraction tools to facilitate the task (e.g., [18, 3, 10, 9, 8, 24].) Many of the protocols that have been proposed and studied (e.g., [19, 29, 27, 6]) are probabilistic. An obvious question is therefore whether we can combine verification tools of parameterized systems with those of probabilistic ones.

The first such attempt is perhaps [32], where the probabilistic selections are converted to compassion (strong fairness) requirements (in a way that exactly captures P-validity of a certain class of temporal properties) and network invariants [22] are manually crafted to abstract the protocols. While this method is sound, as other methods based on network invariants, it is often difficult to apply.

An alternative approach to dealing with such protocols is to replace the probabilistic choices in a protocol with a finite string of "good" random selections. This is based on the observation that for every finite string σ of probabilistic choices, each truly random infinite string of coin flips (made by the same probabilistic coin tosser) is bound to have infinitely many occurrences of σ (in other words, the measure of paths that have σ infinitely many times is 1). To apply this idea, the proof of P-validity is replaced by "regular" liveness proof, where the the random coin flips are replaced by non-deterministic ones that, infinitely many times, at arbitrary points, produces the particular good string of flips. This method, in effect, removes the probabilistic elements of a protocol making it purely non-deterministic, and amenable to other methods for verification.

This method was applied in [4] where the recurring finite string was termed a "planner" (later a "strategy") and might depend on the parameter value or system state. An alternative formulation is called a "pattern" [11]. A pattern is a *non-repeating* infinite sequence of strings that is independent of the parameter value or system state. The two approaches differ in their pragmatics, but have the same theoretical power.

The power of these methods is that, once a pattern/planner is obtained, one can use "standard" techniques on the resulting non-probabilistic parameterized model checking problem (PMCP) and these are (usually) simpler to apply than network invariants. Yet, finding a good pattern/planner may present an intricate puzzle and requires deep understanding of the protocol. Some impressive progress has been made [20] in automating pattern discovery using automaton learning techniques, but this process may diverge. Moreover the PMCP is itself undecidable, posing an additional challenge.

The need for a planner arises from the need to reason about a *parameterized* Markov Decision Process (MDP). In effect, it represents a parameterized strategy for an infinite game played between a probabilistic player and a non-deterministic adversary. We suggest a novel approach to this type of reasoning. In our approach we *avoid the planner* by reasoning in two steps. First, we *abstract* the parameterized MDP to a finite MDP. The probabilistic choices of this abstraction are drawn not from an independent identically distributed (i.i.d.) random variable, but instead from a *parameterized* Markov chain. That is, the distribution of the random variable at any time is dependent on its history and also on the parameter N – the parameter (say, number of processes) the system depends on. In the second step, we prove properties about infinite behaviors of the Markov chain and transfer these to the finite MDP. At this point, the proof can be completed by ordinary finite-state model checking.

By using abstraction to separate parameterization from nondeterminism, we eliminate the parameterized game and avoid the need for a carefully crafted planner. Moreover, our abstraction allows us to work with probabilities, rather than abstract them away, while still enabling us to effectively reason about the protocol.

Of course, finding good abstractions is an art, and does require understanding of the protocol. We will see using some example protocols, however, that the abstraction approach captures in a direct way our intuition about the correctness of the protocol, namely that there is an underlying random walk that is independent of scheduler choice. The approach provides an alternative in case a planner cannot be discovered or the PMCP cannot be solved. We also think it provides a useful insight into the nature of the protocol.

2 Preliminaries

As a computational model for parameterized bounded-data systems we define *fair probabilistic transition systems*, that are a variant *fair transition system* of [21].

A *probabilistic fair transition system* is described by $\mathcal{D} = \langle V, \Theta, \mathcal{T}, \mathcal{P}, \mathcal{J}, \mathcal{C} \rangle$, with:

- $V = \{u_1, \ldots, u_n\}$ — A finite set of typed *system variables*. A *state* s of the system provides a type-consistent interpretation of the system variables V, assigning to each variable $v \in V$ a value $s[v]$ in its domain. Let Σ denote the set of all states over V. An *assertion* over V is a first order formula over V. A state s satisfies an assertion φ, denoted $s \models \varphi$, if φ evaluates to T by assigning $s[v]$ to every variable v appearing in φ. We say that s is a φ-state if $s \models \varphi$.
- Θ — The *initial condition*: An assertion characterizing the initial states. A state is called *initial* if it is a Θ-state.
- $\mathcal{T}$ — A set of labeled transitions. Every transition $\tau \in \mathcal{T}$ is an assertion $\tau(V, V')$ relating the values V of the variables in state $s \in \Sigma$ to the values V' in an S-*successor* state $s' \in \Sigma$. Given a state $s \in \Sigma$, we say that $s' \in \Sigma$ is a τ-*successor* of s if $\langle s, s' \rangle \models \tau(V, V')$ where, for each $v \in V$, we interpret v as $s[v]$ and v' as $s'[v]$. Let $En(\tau)$ denote the assertion $\exists V'.\tau(V, V')$ characterizing the set of states in which τ is enabled. We refer to the transition whose label is ℓ by τ_ℓ.
  A system is *deterministic* if for every $s \in \Sigma$, there is at most one $\tau(s) \in \mathcal{T}$ such that $s \models En(\tau)$.

- $\mathcal{P}$ — A set of labels of *probabilistic selections*. Each probabilistic selection is a tuple $P = \langle \tau_1(V, V'), \ldots, \tau_k(V, V') \rangle$ of transitions that are enabled in the same states, *i.e.*, for all i, j, $1 \leq i, j \leq k$, $En(\tau_i) \equiv En(\tau_j)$. A selection P indicates that $\tau_1, \ldots, \tau_k$ are the the possible outcomes of P, which we refer to as *modes*. The modes of the transitions are associated with probabilities (that sum to 1). We assume that there exists some $\epsilon > 0$ such that all probabilities are greater than ϵ, thus, they are all bounded from below. A *mode label* is of the form ℓ^i where ℓ is the label of a probabilistic selection P and $i \in 1 \ldots k$ is the index of a mode of P. We also assume that each transition in $\mathcal{T}$ appears in at most one probabilistic selection. In other words, the set $\mathcal{T}$ can be viewed as partitioned, with each transition belonging to some probabilistic selection or appearing as a singleton set (and then is not a member of P for any $P \in \mathcal{P}$.)
  If $\mathcal{P} = \emptyset$ then the system is a standard fair transition system.
- $\mathcal{J}$ — A finite set of *justice* (weak fairness) requirements. The justice requirement $J \in \mathcal{J}$ is an assertion, intended to guarantee that every computation contains infinitely many J-states (states satisfying J).
- $\mathcal{C}$ — A finite set of *compassion* (strong fairness) requirements. Each compassion requirement is a pair of assertions, $\langle p, q \rangle$ intended to guarantee that every computation that has infinitely many p-states also has infinitely many q-states.

Fix a system $\mathcal{D} = \langle V, \Theta, \mathcal{T}, \mathcal{P}, \mathcal{J}, \mathcal{C} \rangle$ and let L be the set of all transition labels (*i.e.* labels of non-probabilistic transitions) and mode labels of $\mathcal{D}$. We consider infinite trees, whose nodes are labeled by states and whose edges are labeled by elements of L. A tree T is a *computation tree* if the following conditions hold:

- *Initiality* — The root of the tree, n_0, must be labeled by a Θ-state;
- *Consecution* — For every tree node n labeled by s whose children are $n_1, \ldots, n_a$ labeled by $s_1, \ldots, s_a$ respectively, exactly one of the following holds:
  1. $a = 1$ and the edge from n to n_1 is labeled by a transition ℓ such that ℓ does not label any transition in $\mathcal{P}$ and $(s, s_1) \in \tau_\ell$;
  2. there exists some $P = \langle \tau_1(V, V'), \ldots \tau_a(V, V') \rangle \in \mathcal{P}$ whose label is ℓ, such that for every $i = 1, \ldots, a$, the edge (n, n_i) is labeled by ℓ^i and $\tau_i(s, s_i)$;
- *Justice* — For every justice property $J \in \mathcal{J}$, every infinite path in T has infinitely many nodes labeled by J-states;
- *Compassion* — For every compassion property $\langle p, q \rangle \in \mathcal{C}$, every infinite path in T has either infinitely many nodes labeled by q-states or only finitely many nodes labeled by p-states.

We say that a computation tree is *admissible* if the measure of the set of just and compassionate paths in the tree is 1 (according to the standard definition of the measure of set of paths in a trees, as in [31, 28]). We say that a temporal property φ is *P-valid* in a computation tree if the measure of paths in the tree that satisfy φ is 1. Similarly, φ is *P-valid over* $\mathcal{D}$ if it is P-valid over every admissible computation tree of $\mathcal{D}$.

Note that when $\mathcal{D}$ is non-probabilistic, that is, when $\mathcal{P}$ is empty, then the notion of P-validity over $\mathcal{D}$ coincides with the usual notion of validity over $\mathcal{D}$.

2.1 Parameterized Systems

In this work we are interested in parameterized systems, where there is a *parameter* N such that the system can be instantiated over any $N \geq 1$ processes. The description of components of the system may depend on the parameter. *Uniform verification* of such system allows one to verify all values of the parameter ($N \geq 2$) at one fell swoop [30].

For a running example we take a variant of the Israeli-Jalfon mutual exclusion protocol of [14] for a token ring. The system is *stable* when there is a single token in the system. At times the system may become unstable, and the protocol guarantees that it eventually stabilizes. This problem is one of many that admit no symmetric (where each process follows the same protocol) solution that does not involve some randomization. (The impossibility proof is similar to that in [19] for the case of the dining philosophers.)

In the Israeli-Jalfon solution, the processes are arranged in a bi-directional ring, where each process has a "left" and a "right" link. When a token holder is scheduled, it flips a coin and decides whether to send its token to the left or to the right. If a two tokens "collide", that is, a process holds them both, one of them gets eliminated.

Assume the N processes are arranged in the ring as $P_0 \ldots, P_{N-1}$. For $0 \leq i \leq N-1$, we denote by $i \oplus 1$ the number $(i+1) \bmod N$, and by $i \ominus 1$ the number $(i-1) \bmod N$.

Presented as a parametrized (by N, the number of processes) protocol, the system's variables can be an array *token* of N booleans, such that $token[i]$ denotes that process i has a token. The initial condition $\Theta(N)$ is that the number of tokens in the system is positive, that is, $\bigvee_{i=0}^{N-1} token[i]$. There are three types of transitions in the system. One of a non-token holder that does nothing, one of a token-holder that moves its token to the right, and one of the token-holder that moves its token to the left. More formally, for each i, we have three transitions:

$$\mathcal{T}_1(i) : \neg token[i] \ \wedge\ \bigwedge_{j=0}^{N-1} token'[j] = token[j]$$
$$\mathcal{T}_2(i) : token[i] \ \wedge\ \neg(token'[i]) \ \wedge\ token'[i \oplus 1] \ \wedge\ \bigwedge_{j=0, \neq i, i\oplus 1}^{N-1} token'[j] = token[j]$$
$$\mathcal{T}_3(i) : token[i] \ \wedge\ \neg(token'[i]) \ \wedge\ token'[i \ominus 1] \ \wedge\ \bigwedge_{j=0, \neq i, i\ominus 1}^{N-1} token'[j] = token[j]$$

and the transition relation, $\mathcal{T}(N)$, is the union of all three types over all $i = 0, \ldots, N-1$. For each i there is a probabilistic selection $P(i) = \langle \mathcal{T}_2(i), \mathcal{T}_3(i) \rangle$, where each mode has probability half, and $\mathcal{P}$ includes all $P(i)$'s. Thus, a step of process i is the deterministic $\mathcal{T}_1$ if process i does not hold a token ($\neg token[i]$), or the probabilistic $P(i)$ when process i holds a token.

Finally, for every process i there is a justice property $J(i)$. The goal of $J(i)$ is to guarantee that process i takes infinitely many moves. Since the only variable associated with process i is $token[i]$, and whenever a process i takes a step it doesn't have the token right after it takes the step, it suffices to let $J(i)$ be $\neg token[i]$. The system has no compassion properties.

The self-stabilization property of the protocol is:

$$\phi: \quad \Diamond \Box (\exists i. token[i] \ \wedge\ \forall j \neq i. \neg token[j])$$

of, alternatively, that $\Diamond \Box (\sum_{i=0}^{N-1} token[i] \ = \ 1)$.

2.2 Trading Measure Theory for Fairness

Since the introduction of probabilistic protocols and in the early 1980s, there have been several proposals how to replace reasoning about measure theory in their verification. This is especially easy when studying P-validity (validity with probability 1) with systems where all probabilities are bounded from below, since the actual probabilities can be ignored (replaced, e.g., by uniform distribution over outcomes).

One such approach, initiated by Pnueli [23], is to add some new fairness constraints that depend on probabilistic choices, such that validity over all fair paths implies P-validity in the measure theoretic sense. In other words, a finite set of sets of paths with measure 0 is excluded, leaving the system with a set of paths whose measure is 1. This remaining set of paths can be described using fairness constraints, allowing the property to be verified using standard, non-probabilistic, verification methods.

There are variants among the fairness constraints that have been proposed. On the two ends of the spectrum are *extreme fairness* of [23] which is sound but incomplete. to α*-fairness* of [28] which is sound and complete for finite-state systems. In between there are variants that are all sound. Here we are dealing here with parametrized systems, for which even the simplest properties are undecidable ([2]), thus we restrict sound methodologies, for which we choose one that is in between the spectrum that was termed γ*-fairness* in [4] where γ stands for "global".

A computation $\sigma = s_0, s_1, \ldots$ of a probabilistic transition system is γ-fair if it is just, compassionate, and, in addition, for every state s and probabilistic selection $P = \langle \tau_1(V, V'), \ldots \tau_k(V, V') \rangle$, if for infinitely many i's, $(s_i, s_{i+1}) \in \tau_j$ for some $j \in [1..k]$, then for every $j \in [1..k]$, there are infinitely many i's such that $s_i = s$ and $(s_i, s_{i+1}) \in \tau_j$. In other words, γ-fairness requires that if some probabilistic selection is taken infinitely many times from the same state, then each mode of it is taken infinitely many times from that state. The notion of γ-fairness is sound:

Theorem 1. *If all γ-fair computations of the system satisfy a temporal property ψ, then ψ is P-valid over the system.*

One of the implications of the study of γ-fairness, which was first implemented in [4], is that instead of verifying a probabilistic system, one can verify a similar, non-probabilistic system, such that if the latter satisfies a property, this property is P-valid over the former. The "similar non-probabilistic" system is obtained as follows:

1. All probabilistic selections are replaced by non-deterministic selections;
2. The system is (synchronously) composed with a "decider"[3] (which, as stated in the introduction, has several names including planner, strategy, and pattern) given a state of a system and a probabilistic selection, determines a mode that is to be taken;
3. The system alternates between two modes. In one it replaces all probabilistic selections by non-deterministic selections, that is, each $P = \langle \tau_1(V, V'), \ldots, \tau_k(V, V') \rangle$ is replaced by a non-deterministic choice between $\tau_1, \ldots, \tau_k$. In the other mode, each probabilistic selection is replaced by the mode determined by the decider;

[3] whose proper name is W

4. At arbitrary (non-deterministic) points, the system is in the second mode for a pre-determined finite sequence of probabilistic selections which have to eventually occur according to the fairness requirements.

For example, consider the Israeli-Jalfon mutual exclusion protocol above. There are many ways to define a decider. For example, consider the following decider that attempts to make location 0 the ultimate token holder. Assume that N is odd. Let location in the range $[1, \frac{N-1}{2}]$ be *right locations* and locations in the range $[\frac{N+1}{2}, N-1]$ be *left locations*. The decider proceeds in (at most $\frac{N-1}{2}$) phases, at each it focuses on the token holder with the highest index on the right, and the lowest index on the left. When each of these are scheduled, the decider makes it move the token to the left (resp. right). That is, when $P(i)$ is to be taken, then if i^* is the maximal index of the token holder on the right ($1 \leq i \leq \frac{N-1}{2}$), the decider awaits until i^* is scheduled and then chooses the $\oplus$ mode for it, and if i^* is the minimal index of a token holder on the left, the decider chooses the $\ominus$ mode for i^*. Thereafter, as long as the decider is active, $\neg token[i^*]$ holds. All other modes can be chosen non-deterministically.

At the end of each phase the decider reduces the distance between the extreme right, or extreme left, token holder and 0, after no more than N phases there will be a single token at position 0.

In essence, this methodology calls for replacing probabilistic choices with additional, yet somewhat restricted, non-determinism and to treat the verification of probabilistic parameterized system as if they are "normal" non-deterministic parametrized systems which are, in turn, usually verified using abstraction (most often data abstraction or control abstraction, see [18].)

Crafting a decider is often tricky (as above example demonstrate), and several methods have been proposed to accomplish that in more automated fashion.

Here we pursue a different approach for verification of probabilistic parameterized systems: we propose to abstract a probabilistic parameterized system into a probabilistic non-parameterized one and then directly verify it using finite-state methods for P-validity.

3 Probabilistic Abstraction

The main two methods to verifying large systems (that are too large to be handled by model checking) are *data abstraction* and *control abstraction*. Roughly speaking, the former directly abstracts the components of the transition system, while the latter constructs, in essence, a new system that is shown to implement the original one. In both these methods, the abstraction are constructed such that every property that is valid in the abstracted system is also valid in the original (concrete) system. (We consider both regular model checking and the method of invisible invariants to fall under the former category since, in essence, if successful, they provide an abstraction of a system components.)

A detailed formal description of the two abstraction methodologies is in [18]. Both methods can be easily (though cumbersomely) applied to probabilistic systems, where

the probabilistic selections are kept as a tuple of transitions with the correct probabilities assigned.

In the following sections we give several examples of applications of the strategy advocated here. In all but one (namely, Rabin's Choice Coordination Protocol in Section 4) we use counter-abstraction, and in this section we elaborate on its probabilistic version. We emphasize that other (control or data) abstractions can be used, after extending them to the probabilistic case. Note that "counter abstraction" is traditionally used to abstract each state according to how many processes are in each location, while our use of the method here is more liberal and not restricted to locations.

The current methods to verify that φ is P-valid over probabilistic parameterized system, first use a decider to convert the probabilistic choices to non-deterministic transitions, and then apply some abstractions to the resulting system. The decider (together with the fairness assumptions) guarantee that if φ is valid over the new system, it is P-valid over the original one. Consequently, if φ is valid over the abstract system, it is P-valid over the original one.

Appendix A provides a detailed example of extending [18] to probabilistic systems. Control abstraction can be similarly defined.

3.1 Finite Counter Abstraction for Probabilistic Systems

The term *counter abstraction* is often used to describe an abstraction that replaces a state of a parameterized system (that usually consists of the state of each process together with values of global information) with a vector that assumes a finite set of possible local states, and replaces the vector of states of each process by the count of number processes in each state. Thus, the vector $[0012]$ indicating that processes 0 and 1 are in state 0 and processes 2 and 3 are in states 1 and 2 respectively, will be abstracted into $[2, 1, 1]$ indicating that there are 2 process in location 0 and one each in states 2 and 3.

Counter abstraction is one of the oldest abstraction methods of parameterized systems and has been used for decades to establish safety properties of such systems. The abstraction becomes finite when instead of using the full range, say $[1..N]$, to the counters, one can partition the range into finitely many segments and replace each counter with its appropriate segment. In [25] we observed that in practice often it's enough to partition the range into 0, 1, and "more than 1". A formal definition is in Appendix A.

There we also noted that to obtain the abstraction of the justice properties we need to augment the states with auxiliary information (recording which transition entered the current state) which is then used to define new compassion properties to replace the trivial justice. In retrospect, this could have been accomplished directly by some annotation, which we call here *marking*, on transitions, and to state the compassion properties in terms of this annotation.

For example, consider the counter-abstracted state $[2, 1, 1]$ above. If we abstract the state into finite counters in the range 0 (for none) and 1 (for more than one), we obtain the abstract state $[1, 1, 1]$. Assume now that processes in location 0 can always transition into location 1. Thus, from the (non-finite) abstracted state $[2, 1, 1]$ the system can reach state $[1, 2, 1]$ and then $[0, 3, 1]$. Consequently, the (finite-) counter abstracted state $[1, 1, 1]$ can lead to $[1, 1, 1]$ **or** $[0, 1, 1]$. This non-determinism in the abstraction is not surprising. Now suppose that there is a fairness assumption that requires that for

every process i, i cannot remain forever in location 0. Thus, we have N (where N is the number of processes) justice properties that require $\neg\pi[i] = 0$ where $\pi[i]$ is i^{th} location counter. Since the counter abstraction doesn't offer us means to track any particular process, the only possibly abstraction of these N justice properties is T. However, the role of justice is to rule out "bad" schedulers, and a T justice rules out nothing.

In fact, we know that, as long as no process joins location 0, the transition that keeps the system in the abstracted state $[1, 1, 1]$ can be taken finitely many times, namely, as many times as the number of (concrete) processes in location 0. The solution of [25] to this problem is to add a new *compassion* property that requires that if a process leaves location ℓ infinitely many times, then a process enters location ℓ infinitely many times. To record which locations are left/entered, [25] annotates the (abstract) states with *to* and *from* variables.

We follow the idea, however, place the marking on the transitions. So that the transition of a process in location from the (finite counter) state $[1, 1, 1]$ into itself is marked by $m = (-, +, 0)$ denoting the the first location was decremented, the second incremented, and the third is unchanged. This is also the marking of the transition from $[1, 1, 1]$ into $[0, 1, 1]$. With each transition we associate a compassion property that states that for every coordinate, if a computation has infinitely many $+$ marking on that coordinate, it must have infinitely many $-$ marking on that coordinate, and vice versa. In fact, it is often the case that a transition in a (finite) counter-abstracted system has an *escape* mode that is taken when a pointer is depleted. Here the transition into $[0, 1, 1]$ is such an escape for the outgoing transition from $[1, 1,]$ marked by m. We describe the marking in Section 3.3.

3.2 P-validity on abstract system

If we manage to successfully counter-abstract a system so that it defines, say, an absorbing Markov Chain with the absorption corresponding to the goal state we wish to verify, then the abstraction provides us with the required proof.

Example 1. Consider the Israeli-Jalfon protocol. The decider in the previous section is a refinement of some intuition to verify the protocol, namely, that if the length of the maximal chain of token-less processes monotonically increases then eventually there will be a single token holder. The decider arbitrarily decided on 0 as the convergence point, and its actions are designed so that it guarantees the single token to end up there. Every decider implies some possible abstraction. The decider we used implies a counter abstraction where the counter may be, for example, the sum of distances from the end-point processes to position 0. Yet, we do not, and don't need to, fully define the decider, or a more precise planner/pattern, rather we use our somewhat vague intuition to abstract the system. This is true in general: Crafting a correct decider requires a deep understanding of the protocol, while our abstraction, which maintains the probabilistic selection, is considerably less detailed.

In this case we can reason as follows. The protocol attempts to reduce the number of tokens, so one counter can store the number of tokens. The only way to reduce the number of tokens is to have collisions, and the closer tokens are the more collisions there are. Hence, the sum of distances between tokens also seems to be a candidate

for a counter. Since the closer tokens are, the faster there are collisions, it makes sense to "reward" the system when this distance is small. To this end, we assign distances between tokens non-linear weights, such that the less distance there is between adjacent tokens are the less weight they carry (in other words, guaranteeing that "less is more.")

In more details, when there are more than one token in the system, define, for every position i:

$$d(i) = \begin{cases} 0 & \neg token[i] \\ \text{minimal hops \# of from } i \text{ to collide with a token} & token[i] \end{cases}$$

and let $w(i) = \log(d(i))$ when $d(i) \neq 0$ and 0 otherwise. The counter abstraction is then $(\sum_{i=0}^{N-1} token[i], \sum_{i=0}^{N-1} w(i))$.

Consider, for example, a state where the token holder process i is scheduled and let j_1 and j_2 the closest token holders on both its sides. That is, in the ring there is a sequence:

$$token[j_1], \underbrace{\neg token[j_1 \oplus 1], \ldots,}_{\text{no token holders}} token[i], \underbrace{\neg token[i \oplus 1], \ldots,}_{\text{no token holders}} token[j_2]$$

The case that $j_1 = j_2$ and similarly to the case that $j_1 = i \ominus 1$ or $j_2 = i \oplus 1$ can be easily handled. Assume it's none of these cases, i.e., that $j_1 \neq j_2$, $\neg token[i \oplus 1]$, and $\neg token[i \ominus 1]$. To avoid modulo reasoning, we assume, without loss of generality, that $0 \leq j_1 < i < j_2 \leq N-1$ and that $(i - j_1) \leq (j_2 - i)$. Denote $d = d(i)$. We now consider the new weights once i moves its token to the left:

1. $w'[i] = 0$ becomes 0 since $\neg token'[i]$;
2. $w'(i-1) = \log(d-1)$ (since the $d'(i+1) = d-1$);
3. $w'(j_1) \leq w(j_1)$: The move may decrease $d(j_1)$ and cannot increase it;
4. $w'(j_2) \leq \log(d+1)$: If $d(j_2) = d$, the move *may* increase $d(j_2)$ to $d+1$. Otherwise, $d(j_2)$ remains intact.
5. No other weight is impacted.

It follows that $w(i)$ is replaced by the smaller $w'(i-1)$, but that when $d(j_2) = d$, $w(j_2)$ may increase. In this case we have:

$$w(i)+w(j_2) = 2\cdot\log(d) = \log(d^2) > \log(d^2-1) = \log(d-1)+\log(d+1) = w'(i-1)+w(j_2)$$

(Any other concave function can replace log[4].) It follows that a transition of a token holder has a positive probability (bounded from below by $\frac{1}{2}$) to bring the system into a state where $(\sum_{i=0}^{N-1} token'[i], \sum_{i=0}^{N-1} w'(i)) \prec_{lex} (\sum_{i=0}^{N-1} token[i], \sum_{i=0}^{N-1} w(i))$. This also induces a potential ranking function on states, such that every (probabilistic) transition has a mode that reduces the rank. The rank is well founded using lexicographical ordering among its coordinates. The first coordinate is in the range $[1..N]$. For a given N, the second coordinate is any value that can be obtained summing weights of k, $2 \leq k \leq N$ distances whose sum is bounded by $N-k$. This value can be computed, but we don't have to compute it. All that we need is to know that it can take on finitely many non-negative values and therefore $\leq$ is well-founded over it.

[4] We thank Xinhua Zhang for pointing this out, as well as suggesting the log function, to us

Together with a scheduler, the (non-finite) counter abstraction defines a 2D-random walk, where the x-coordinate is in the range $[1..N]$ (number of tokens) and the y-coordinate is some finite range (for weights), that is bound to reach $x = 1$. A move along the x-axis can only decrement x. A move along the y-axis can increase or decrease y, but there is always a mode (positive probability) of reducing y, and, eventually, x. From here we can use a finite counter abstraction, or just reason directly on the random walk.

Once we realize that every token-holding process has a mode that makes progress in the chain (until the x-coordinate reaches 0), we can abstract away the non-determinism and consider a single transition that can always progress towards the goal, where we don't need to quantify progress (which is nondeterministic and hard to compute), and one that may increase the y-coordinate.

3.3 Transition Marking

Assume the state is abstracted into k counters. Each transition will then be marked with a k-tuple over $\{=, -, +\}$. Consider a (concrete) from location ℓ into location ℓ'. In the abstract system, the coordinate corresponding to ℓ is marked with $-$, and the one corresponding to ℓ' with $+$. The associated compassion-like requirement then will change to requiring that for every coordinate, if infinitely many transitions have $+$ marking on the coordinate, then infinitely many transitions have $-$ marking on that coordinate, and vice versa.

For probabilistic selections the markings are similar. There, however, one needs to add the escape transition to every mode of the a probabilistic selection.

Recall that obtaining counter abstraction is done in two steps. In the first the system is counter abstracted into unbounded counters, and in the second the unbounded countered are bounded into ranges. It is often easier to mark the transitions on the unbounded counter system, where the marking does not add any information about the system and where the escape transitions do not exist. However, once they appear in the unbounded countered system, it's trivial to construct them for the finite state counter abstracted system.

Example 2. Consider the Israeli-Jalfon protocol. In the previous example we presented an unbounded counter abstraction of the system that describes a 2D random walk. From each point, each probabilistic selection may lead to a "better" state: one where the x-coordinate is lowered, or where the x-coordinate remains the same and the y-coordinate is lowered. In fact, we can't always exactly compute the possible values of the y-coordinate, but we know that for each N there are finitely many such values. Moreover, the y-coordinate can be reduced (or increased) by many values, depending on the exact configuration and the process that is scheduled. The power of the methodology we advocate here is that we don't need to know these details. We can abstract the system so that it includes two states, (m, m) denoting that there are more than one tokens and the sum of the weights is "whatever", and $(1, m)$ denoting there is a single token. From (m, m) there is a single probabilistic selection with two modes:

1. The first mode corresponds to a good selection that reduces the current location. It is marked by a non-deterministic choice $(=, -)$ or $(-, ?)$ where $?$ abbreviates $+$, $-$, or $=$ (all three possibilities), with an escape mode to $(1, m)$;
2. The second mode corresponds to a selection that may, like the previous case, move towards the goal, or move away from the goal, but only on the y-coordinate. Note, e.g., that when a token holding location is in the midst of a chain of token, either move is good as it reduces the number of tokens. When there are no immediate token-holding neighbors, a move of a token away from the closest token may still reduce the sum of the weights, i.e., maintain x and reduce y. This mode can be marked with a non-deterministic selection between $(=, ?)$ and $(-, ?)$. The first, $(=, ?)$, may at time cover moves where the y coordinate is increased (while the x one remains intact, but then, the transition may always choose the good mode described in (1).

As we noted in the previous example, this is clearly an absorbing Markov Chain and it may be easier to verify it reaching stabilization using this observation.

Going back to our transition marking, a random walk that always has a non-zero probability of decreasing will escape any lower bound infinitely often, even if we allow infinitely many increases. Let us then put the probabilistic selections and the non-probabilistic transitions into three classes with respect to counter i (treating the non-probabilistic case as a single-mode selection). A *decreasing* selection is one which has some mode marked $-$. A *maintaining* selection has all modes marked $=$. An *increasing* selection is anything else. With this terminology, we can now give a compassion constraint on marked probabilistic systems with k counters, each counter i with a lower bound V_i:

> For every $i = 1, \ldots, k$, a computation that has infinitely many decreasing transitions with respect to counter i and only finitely many increasing ones must take infinitely many escapes with respect to V_i.

In other words, the set of runs in which we go on an infinite random walk without infinitely escaping some lower bound can be eliminated as having zero measure. The exclusion of a set of zero measure doesn't change the measure of the remaining set of computations (which is 1). The key difference in the probabilistic case is that a probabilistic transition can be treated as decreasing *even if it has both* $-$ *and* $+$ *outcomes*.

This is of course not the only set of paths we remove, since we also ignore all paths that are not extremely fair, i.e., those that take the same probabilistic selection from the same state infinitely many time and always choose the same mode. This too is a measure 0 set. Thus, we can use fairness properties to identify a measure 1 set of paths, and can use any finite-state model checker to verify the resulting system.

3.4 Summary

To summarize the method, we take the following steps:

1. Choose appropriate counters and construct an infinite counter abstraction of the system.

2. Abstract to finite state, keeping marks on the transitions to preserve the counter directions.
3. Strengthen the finite abstraction by adding compassion constraints that derive from extreme fairness and treating the counters as a random walk.
4. Use finite-state model checking to prove correctness.

Notice that in this method, we never prove properties about a non-deterministic parameterized system. The only parameterized system we prove properties about is a Markov chain (and a generic one at that). This is in contrast to the planner/pattern approach in which we convert a parameterized MDP to a non-deterministic parameterized system, which we must then prove correct by some means.

While we used a result about random walks to remove a set of behaviors of zero measure, there is no reason in principle why we could not consider sets of finite measure. In other words, we could use the same method to prove properties that hold with some quantitative probability bound less than one.

4 Rabin's Choice Coordination

In [29] Rabin gave a protocol where N processes have to choose one of 2 cells by means of writing values in a given range onto the cells, and a special value denoting that a cell is "chosen". The goal of the protocol is to guarantee that with a high probability (depending on the cardinality of the set of values that can be written onto a cell) a single cell will be "chosen" while it is never the case that both cells are chosen. We give a variant of the protocol here and use our methodology to prove its correctness.

Assume two cells, *cell0* and *cell1*, each can take on a value in $\{0..M\} \times \{0, 1\}$. Unless the cell's value is $(0, 1)$, the first coordinate is its *phase* and the second is its *value*. Both cells are initialized to $(0, 0)$. There are $N > 0$ processes, each pointing to some cell and having three states, *playing*, indicating it is still trying to choose a cell, *gaveup* (indicating it has given up choosing a cell), and *decided* indicating it chose a cell. Once a process gives up or decides, it remains pointing to the same cell. Otherwise, it may change the value of the cell it is pointing to, and switch to the other. The goal is to have one cell chosen by all N processes, or all processes giving up. A chosen cell has a value $(0, 1)$.

A playing process "remembers" the value of the last cell it has seen. If it sees a larger value, it leaves that cell's value unchanged (and switches cells), if it sees a smaller value, it choses that cell, by setting the cell's value to $(0, 1)$, and remains there in a "decided" state. If it sees an equal value to what it last saw and the phase of the cell is less then M, it increments the cell's phase and probabilistically chooses a value (in $\{0, 1\}$) for it. If, however, the phase is M, it gives up and remains on the cell. Recall that a playing process always switches cells after reading one, and that a process that has decided or given up remains pointing to the same cell.

4.1 Reducing to a two-process system

To prove a bound on the probability of successful termination, it will help to simplify the problem by reducing it to two-process systems. We can show that a two-process

system simulates an N-process system, in the non-probabilistic sense. More precisely, for any sequence of coin flips and for any schedule of N processes, there is a schedule of 2 processes that performs the same sequence of operations on the cells.

We used TLV to obtain the proof for several instantiations of N (we stopped at 9 out of boredom rather than state explosion) and are working on getting the proof for general N in IVY. We can also construct a manual proof: roughly speaking, we track the two most advanced processes in the system. Initially these are any two processes such that if there are processes on both sides, we chose one from each side. If one process decides or gives up, this process is one of the two. Otherwise, the cells contents are (k_0, b_0) and (k_1, b_1) where $K = \max(k_0, k_1) > 0$. If $k_1 = k_2$. If there exists a process who saw both current values, we choose this process. If there are more than one, we choose both. If there are now two such processes, we choose those who last wrote on the cells. (There must be two of them.) Otherwise, assume without loss of generality that $k_0 > k_1$. We choose the process who last wrote on *cell0*, and for the second process we choose anyone who saw *cell0* before the update (when its value was that of the current *cell1*). If no such process exists, any would do.

For the 2-process system, we proved, using invisible invariants, (1) and (2) below. Note that (1) has to do with the cell values (thus uses the parameter M) while (2) is only about the processes. To prove (3) we used a simple conversion of liveness to safety, from where we used invisible invariants.

$$\Box(\neg(\mathit{cell0} = (0,1) \wedge \mathit{cell1}(0,1))) \tag{1}$$

$$\forall i \neq j.\ \ \Box(\neg(\mathit{state}[i] = \mathit{decided} \wedge \mathit{state}[j] = \mathit{gaveup})) \tag{2}$$

$$\forall i.\ \ \Diamond(\mathit{state}[i] = \mathit{decided} \vee \mathit{state}[i] = \mathit{gaveup}) \tag{3}$$

The last property only holds under fairness, that is, when each process takes infinitely many steps.

These are all properties of a non-probabilistic system and make no assumptions on the random choices.

4.2 Showing successful termination with high probability

We first show that once the contents of both cells are equal and the processes are on different sides, both having observed a value at the current phase of the cells, eventually (as a matter of fact, within 4 steps) one of the following occurs:

1. the phase is M and then one processes gives up, which, by (2) and (3) implies that they eventually both do;
2. at least one process decides, which, by (1) and (3) mplies that eventually both decide on the same cell;
3. a similar state is reached, only with a higher phase number.

Note: When both processes are on the same side and the cells have the same content with phase $< M$, then it is easy to show (as well as follows from a portion of Fig. 1), that eventually the cell on this side has its value incremented and is chosen.

Formally, we have:

Lemma 1. *For every process i, let $s_l_p[i]$ denote that the last cell process i saw has phase K which is the maximum of the two phases stored in cells. Then, for every k, $0 \leq k < M$,*

$$\begin{array}{l} \mathit{phase}[0] = k \wedge \mathit{cells_eq} \ \wedge \ s_l_p[1] \ \wedge \ s_l_p[2] \ \wedge \ \mathit{side}[1] \neq \mathit{side}[2] \ \Longrightarrow \\ \left(\begin{array}{l} \mathit{phase}[0] = k+1 \ \wedge \ \mathit{cells_eq} \ \wedge \ s_l_p[1] \ \wedge \ s_l_p[2] \ \wedge \ \mathit{side}[1] \neq \mathit{side}[2] \\ \vee \neg \mathit{playing}[1] \ \vee \ \neg \mathit{playing}[2] \end{array} \right) \end{array}$$

(The formula $p \Longrightarrow q$, read "p entails q, is the *progress* property $\Box(p \rightarrow \Diamond q)$.)

Proof. Since a picture is worth a thousand words, we provide a proof by picture in Fig. 1, where *cells_eq* denotes *cell0* $=$ *cell1*. The dashed boxes denote states. The labels on transitions denote which process is scheduled under which condition. E.g., "$k < M$ any process P_i" indicates when $k < M$, either P_1 or P_2 can lead into the state denoted by the transition.

The states enclosed by dashes are the goal states: the top left denotes that one process gives up, the bottom right denotes that one process decides, and the one in the middle right denotes that the system returns to the state it started with, only with a higher k.

The random coin tosser is denoted by the hand flipping a coin. When the coin flip is used (when scheduling the process who was not last scheduled), with probability $\frac{1}{2}$ the first goal (of returning to the initial state, with a higher phase) is attained, with with probability $\frac{1}{2}$, within 3 steps a one of the processes reaches a decision.

Rather than showing termination, our goal is to show decision with probability $\geq 1 - 2^{-M}$.

Fig. 1 gives us means to abstract the system. Every state where no decision is made and no process gives up can be abstracted to either right-hand-side of Lemma 1 or to the (simpler) case where both cells are equal, the phase is k, both $s_l_p[1]$ and $s_l_p[2]$ hold, but $\mathit{side}[1] = \mathit{side}[2]$. We focus on the first case and abstract it by a counter that equals the phase of the cells, say k. From Fig. 1 it follows that if $k < M$ and the processes are scheduled P_1, P_2 or P_2, P_1, then with probability $\frac{1}{2}$, then the system reaches the abstract state $k + 1$. In all other cases, it reaches a decision. When $k = M$ it reaches a deadlock (giving up) state.

In the (finite) counter abstracted system we have a self loop on the state k with a $+$ annotation on the mode, with another mode leading to a decision. When k is incremented sufficiently many times, it reaches M, in which case the only transition leads to a giving-up state. Since the counter is initialized to 0, after M increments it reaches M. It thus follows, from the counter-abstracted system, that the probability of reaching a giving-up state is bounded from above by 2^{-M}. We thus have:

Lemma 2. *The system reaches a decision with probability at least* $1 - 2^{-M}$.

This is a very different use of counter abstraction than the usual one. It is used to simplify a system and to show that some event occurs with a non-trivial (0 or 1) probability. We gave this example to demonstrate the power of delaying the de-probabilization of coin flips (the decider W accomplishes). In this example, it allowed us to go beyond P-validity.

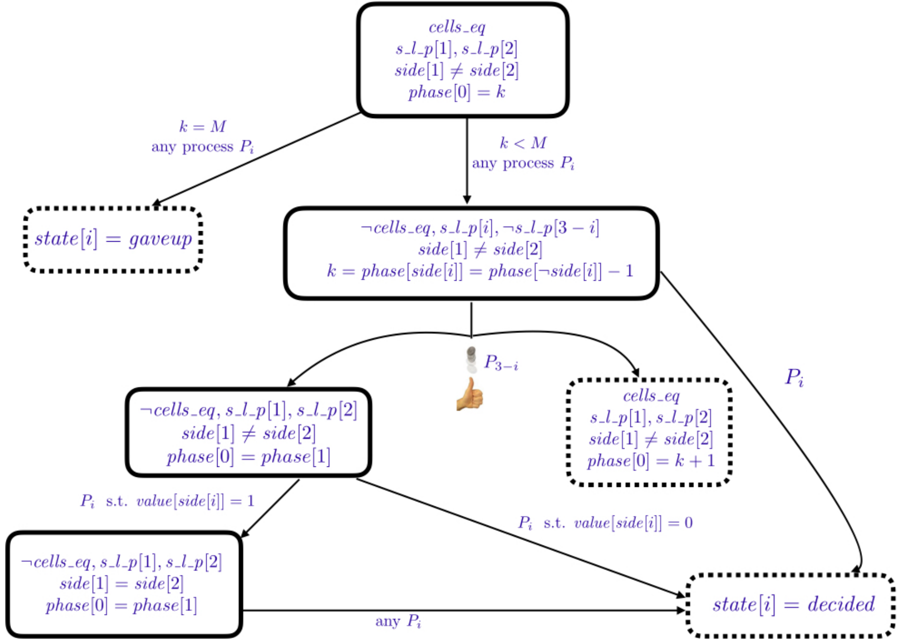


Fig. 1. Proof of Lemma 1

5 Example: A P2P protocol

The work in [16] describes a P2P parameterized probabilistic protocol. We present a variant of it here which we show to be correct using out new methodology.

The protocol assumes a fully connected (clique) of processes, each of which has a weight. At each phase one process is activated. When activated, the process, say i, randomly, with uniform probability, selects one of its neighbors, say j. That is, the choice of i is non-deterministic (and constrained by the obvious justice property: no process can be ignored forever) while the selection of j is probabilistic. Once i chooses j, both "give" the other half of its weight, that is, $w'[i], w'[j] = (w[i] + w[j])/2$. The goal of the system is to have all processes hold weights that are roughly equal.

Let A denote the average of the weights (note the the sum of the weights remains constant throughout the protocol). Let ϵ be some precision requirement. The termination property of the protocol is:

$$\Diamond(\forall i.\ w[i] \in [A - \frac{\epsilon}{2}, A + \frac{\epsilon}{2}])$$

which we want to show is P-valid.

Assume that the range of the original weights are $[L, H]$. Either this range, or a super-range of it, and ϵ itself have to be known in advance in order to make our analysis sound. Without loss of generality, assume that $A - L \geq H - A$ and let $k = \lceil \frac{A-L}{\epsilon} \rceil$. Let $\sigma_{-k}, \ldots, \sigma_{-1}, \sigma_0, \sigma_1, \ldots, \sigma_k$ be $k+1$ consecutive intervals, each of length ϵ, such that $\sigma_0 = [A - \epsilon, A + \epsilon]$. This range obviously covers $[L, H]$. Note that while we chose the intervals with A at the center, this choice of intervals is only for proof purposes and we don't assume A is known in advance. Rather, we prove that once the system operates long enough, it is almost guaranteed to have the weights of all processes in the middle interval σ_0.

Before we proceed, we'll state some properties of the system:

1. Let W be the maximal weight in the system and let i be the single process such that $w[i] = W$. After i is scheduled once, the maximal weight in the system will always be $< W$.
2. Let W be the minimal weight in the system and let i be the single process such that $w[i] = W$. After i is scheduled once, the minimal weight in the system will always be $> W$.
3. Let $j > 0$ be the maximum such that for some i, $w[i] \in j$. Then there is a mode of i's probabilistic selection that, once taken, $w'[i] < j$.
4. Let $j_h > 0$ be the maximum such that for some i_h, $w[i_h] \in \sigma_{j_h}$ and let $j_l < 0$ be the minimum such that for some i_l, $w[i_l] \in \sigma j_l$. Then if i_h is scheduled, there is a mode that if taken, $w'[i_h] \in \sigma_i$ where $j_l < i < j_h$, and similarly for i_l.

As before, some intuition on why the protocol may converge helps in abstracting the system. Here we try to "look" at the movement of processes among the intervals. Obviously (and also from (1) and (2)), processes at the endpoints, once scheduled, move away from the endpoints and will never be replaced — new endpoints will be defined that are closer to A. Eventually the endpoints will be in σ_0 and remain there. The counters may count the number of processes on the extreme intervals, since, once depleted remain so, and we can then focus on more central intervals. To avoid separating the positive and negative sides, we "fold" the domain at the A point. Let the results be $I_0, \ldots, I_k$. That is, $I_0 = \sigma_0$, and for every $I = 1, \ldots, k$, $I_k = \sigma_{-i} \cup \sigma_i$.

At this point we can identify a decider: When activated, start with the highest index j such that for some i, $w[i] \in I_j$ and let Q_j be the set of such processes. When a process in Q_j is scheduled, let it choose the process whose weight is the furthest from it (which may be in the same I_j with an opposite polarity) which moves it out of Q_j. Hence, after at most N activations the highest index j reduces. Unlike other deciders, here we cannot bound the number of times the decider is activated since, as we stated, this depends on the precise instantiation. However, since there are finitely many I's, and at each round of decider we lose one, it is guaranteed that eventually all weights will be in I_0. With this, the decider so defined is way outside the common definition of planner/ strategy/ pattern due to the numerous parameters of the system (range of weights, true average, etc.)

Of course, once we have a decider, counter abstraction is easy. For example, we can use a 2D counter again, where the first coordinate is the maximal j such that $Q_j = \{i : w[i] \in I_j\}$ is nonempty, and the second is the cardinality of this set. The first coordinate

is monotonically increasing, and each step of a process in I_j has a mode that decreases the second coordinate (or, when the latter is 1, it has an escape which decreases the first coordinate), and it can be increased finitely many times by processes outside of Q_j. (This is because the only way into Q_j is for a process p with weight in I_{j_1} to select a process $q \in Q_j$ process. Both may end in Q_j, but their weight will be closer to I_{j_1} than that original weight of q.)

Note that, unlike the decider, the counter abstracted system allows us to ignore all the information that makes it hard/ infeasible for other methods to verify the protocol.

6 Other Examples

We have counter-abstracted (almost?) all the examples of probabilistic parameterized systems that are in the formal methods literature, including another asynchronous token ring protocol ([5]), mutual exclusion protocols ([27], as well as the less cited [6]), and the Ittai and Rodeh's leader election protocol for a ring with a known size (N) ([15]). (We exclude Hermman's Protocol, [13] because of its synchronous nature, though we did verify it using our methodolgies.)

All can be reduced to random walk. In Ittai and Rodeh's leader election protocol, as well as in the mutual exclusion protocols (which are asynchronous and considerably more complicated) there is a notion of "competitors" that flip coins. Those who flip "heads" continue to the next round of the competition, while the others drop out (in the mutual exclusion case, they wait until there are no more competitors.) If nobody flips "heads", all the competitors make it to the next round. Else, only those who flipped "heads" do. Eventually, with probability 1, there is only one competitor who is the winner. While the mechanism of determining whether a competitor in one round is also competing in the next one differs in those three protocols, we can abstract this away and just keep a counter for the number of competitors, which, with a positive probability, decreases at each round. For the mutual exclusion protocols some other counters are needed. For example, in the [27] protocols, there is a notion of a "waiting room" where those who wish to enter the critical section, but either failed to reach the competition in time, or lost the competition, wait. These counters, however, belong to the non-probabilistic part of the system so we need no describe them. See Section **??** for an example of an application of our method to the mutual exclusion protocol of [27].

The group of "usual suspects" of parameterized probabilistic protocols there are some for each local reasoning may suffice. These include both Free Philosophers and Courteous Philosophers of [19] and the leader election in the Firewire protocol [1]. The "free philosophers" algorithm allows a hungry philosopher to randomly choose a side s, wait for its s-chopstick to be available, then if the other chopstick is available, the process eats, else it puts down the s-chopstick and goes back to selecting a side. The "courteous" philosophers algorithm is similar, only there a philosopher Phil is not allowed to pick up a chopstick if its neighbor, with whom Phil shares the chopstick, is also hungry and Phil ate before its neighbor did.

The property whose P-validity one usually want to prove is that if there are some hungry philosophers then eventually one of them gets to eat (obtain both chopsticks), or that if some philosopher is hungry that this philosopher eventually gets to eat. The

P-validity of these protocols can be established by first observing that each hungry philosopher who doesn't eat lifts a chopstick infinitely many times (that is, doesn't starve by indefinitely waiting for one of the chopsticks), a proof that doesn't require probabilistic reasoning. Then, it can be easily shown, using local probabilistic reasoning, that among each two adjacent philosophers, at least one of whom is hungry, one of them gets to eat. In the courteous philosophers case, if one of them eats then the other eventually eats. Thus, once the main (non-probabilistic) claim is established, possibly using counter abstraction or some control abstraction, the P-validity of the protocol can be established using local (2 or 3 adjacent processes) reasoning and doesn't require our methodologies. One can always define some decider to these protocol, but that may be an overkill in the sense that such a decider determines what every process does, while only a few matter. For example, in the Free Philisophers all processes but one can draw "left" while the remaining one draws "right", which will guarantee that there exists a pair whose common fork is available, so that the first process to pick up one fork will be able to eat. (The protocol only guarantees P-validity of "if someone is hungry then eventually someone eats".) But this pair, where the left one draws left the the right one draw right (so once one of them succeeds to pick up a fork it can eat) is essentially local and doesn't need to involve a decider that decides on a number of coin flips which depends on N.

7 Conclusions

Previous approaches to verification of parameterized probabilistic systems have taken the route of reducing P-validity for a parameterized MDP to the ordinary parameterized model checking problem (PMCP) by introducing a pattern or planner [4, 11]. Here we have taken a different route, isolating the parameterized reasoning to a Markov chain. We observed that a number of protocols embed a Markov chain in their state spaces that becomes apparent when the state space is abstracted appropriately. In fact, for several it is the *same* Markov chain. In a way, this argument exposes the underlying common reason why these protocols work, and may actually aid the verifier in finding a suitable planner (if one is still necessary) because of the reduction to a finite state abstraction.

The strategy presented here builds on existing techniques but applies them in a novel way, advocating abstraction of probabilistic parameterized system directly into simpler probabilistic systems rather than first fixing some probabilistic choices (letting the others be nondeterministic). From our experience, this is often simpler and requires less familiarity with the details of a system. Yet, this is a subjective experience and it is difficult to argue whether it is easier in general to discover a pattern/planner or a suitable abstraction. A clear advantage of the abstraction approach is that it avoids the need to solve the undecidable PMCP.

Another possible advantage is that the strategy presented here is also applicable to properties with *quantitative* probability bounds and not just to P-validity. In contrast, selecting a planner instead of abstraction limits the verifier to non-quantitative analysis. We have applied the abstraction approach, for example, to Rabin's Choice Coordination protocol [29]. This seems a promising avenue for future research.

A Data Abstraction for Probabilistic Systems

The method of *finitary abstraction*, introduced in [17], allows to reduce a verification problem into one on a smaller system. The idea behind finitary abstraction (as well as other abstraction methods, all inspired by [7]) is to reduce the problem of verifying that a given *concrete* system $\mathcal{D}$ satisfies its (concrete) specifications ψ, into a problem of verifying that some (carefully crafted, finite-state) *abstract* system $\mathcal{D}^\alpha$ satisfies its (finite-state) abstract specifications ψ^α.

We sketch here the basic elements of the method, and refer the reader to [18] for additional details. Consider a probabilistic transition system $\mathcal{D} = \langle V, \Theta, \mathcal{T}, \mathcal{P}, \mathcal{J}, \mathcal{C} \rangle$. Assume a set of *abstract variables* V_A and a set of expressions $\mathcal{E}^\alpha$, such that $V_A = \mathcal{E}^\alpha(V)$ expresses the values of the abstract variables in terms of concrete variables. For any state assertion $p(V)$, we say $\alpha^+(p)$ is the set of abstract states that abstract emmsome p-state. That is, $\alpha^+(p) = \exists V.\ (V_A = \mathcal{E}^\alpha(V) \wedge p(V))$. Similarly, $\alpha^-(p)$ is the set of abstract states such that abstract *only* p-states, that is, $\alpha^-(p) = \forall V.(\mathcal{E}^\alpha(V) \Rightarrow p(V))$. Both α^- and α^+ can be generalized to temporal formulae [18]. To be conservative, a formula ϕ to be proved is abstracted by $\psi^\alpha = \alpha^-(\psi)$. For example, $(\square(p \rightarrow \diamondsuit q))^\alpha = \square(\alpha^+(p) \rightarrow \diamondsuit(\alpha^-(q)))$.

The abstract system $\mathcal{D}^\alpha$ that corresponds to $\mathcal{D}$ under α is the probabilistic transition system $(V^\alpha, \Theta^\alpha, \mathcal{T}^\alpha, \mathcal{P}^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha)$ where:

- $V^\alpha = V_A$;
- $\Theta^\alpha = \alpha^+(\Theta)$;
- $\mathcal{T}^\alpha = \bigcup_{\tau \in \mathcal{T}} \alpha^{++}(\rho)$, where $\alpha^{++}(\tau) = \exists V, V' : V_A = \mathcal{E}^\alpha(V) \wedge V'_A = \mathcal{E}^\alpha(V') \wedge \tau(V, V')$
- $\mathcal{P}^\alpha = \bigcup_{\langle \tau_1, \ldots, \tau_k \rangle \in \mathcal{P}} \langle \alpha^{++}(\tau_1), \ldots, \alpha^{++}(\tau_k) \rangle$ (recall that all modes of a probabilistic selection share the same preconditions.)
- $\mathcal{J}^\alpha = \{\alpha^+(J) : J \in \mathcal{J}\}$, and
- $\mathcal{C}^\alpha = \{\langle \alpha^-(p), \alpha^+(q) \rangle : \langle p, q \rangle \in \mathcal{C}\}$

Theorem 2. *The abstraction is sound, that is,* $\mathcal{D}^\alpha \models \psi^\alpha \quad \Longrightarrow \quad \mathcal{D} \models \psi$

Proof (Outline). The only part that needs to be added to the proof of [18] is to show that the α-abstraction of every γ-fair $\mathcal{D}$ computation is a γ-fair $\mathcal{D}^\alpha$ computation, which follows from the definitions.

Counter Abstraction. Consider a set V_A of k abstract variables, say $V_1, \ldots, V_k$, each of which can take on values from a finite range that depends on the system's parameter, that is, for each $i = 1, \ldots, k$, V_i evaluates into a value in a finite domain B_i. (Usually, $B_i = [0..g(N)]$ where $g(N) \leq N$.) Let $\mathcal{E}^\alpha$, such that $V_A = \mathcal{E}^\alpha(V)$.

The resulting system $\mathcal{D}^\alpha$ is a *counter abstraction of the system* $\mathcal{D}$.

References

1. IEEE standard for a high-performance serial bus. IEEE standard 1394-1008, 2008.
2. K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6), 1986.
3. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV 2001*, pages 221–234, 2001.
4. T. Arons, A. Pnueli, and L. D. Zuck. Parameterized verification by probabilistic abstraction. In *FoSSaCS 2003*, pages 87–102, 2003.
5. J. Beauquier, M. Gradinariu, and C. Johnen. Memory space requirements for self-stabilizing leader election protocols. In *PODC 1999*, pages 199–207, 1999.
6. S. Cohen, D. Lehmann, and A. Pnueli. Symmetric and economical solutions to the mutual exclusion problem in a distributed system. *Theoretical Comput. Sci.*, 34:215–225, 1984.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.
8. E. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE-17)*, pages 236–255, 2000.
9. E. Emerson and K. Namjoshi. Automatic verification of parameterized synchronous systems. In *CAV 1996*, 1996.
10. E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *POPL 1995*, San Francisco, 1995.
11. J. Esparza, A. Gaiser, and S. Kiefer. Proving termination of probabilistic programs using patterns. In *CAV 2012*, pages 123–138, 2012.
12. S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. In *POPL 1982*, pages 1–6, 1982.
13. T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.
14. A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing (PODC '90)*, pages 119–131. ACM New York, 1990.
15. A. Itai and M. Rodeh. Symmetry breaking in distributed network. In *FOCS 1981*, pages 245–260, 1981.
16. D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 482–491, 2003.
17. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163:2000, 1999.
18. Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 4:2000, 2000.
19. D. Lehmann and M. Rabin. On the advantages of free choice: A symmetric and fully distibuted solution to the dining philosophers problem (exended abstract). In *POPL 1981*, pages 133–138, 1981.
20. A. W. Lin and P. Rümmer. Liveness of randomised parameterised systems under arbitrary schedulers. In *CAV 2016*, pages 112–133. Springer, 2016.
21. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
22. K. L. McMillan. A compositional rule for hardware design refinement. In *CAV 1997*, pages 24–35, 1997.

23. A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In *STOC '83*, pages 278–290. ACM New York, 1983.
24. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS 2001*, volume 2031, pages 82–97, 2001.
25. A. Pnueli, J. Xu, and L. D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 107–122, London, UK, UK, 2002. Springer-Verlag.
26. A. Pnueli and L. D. Zuck. Probabilistic verification by tableaux. In *LICS 1986*, pages 322–331, 1986.
27. A. Pnueli and L. D. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.
28. A. Pnueli and L. D. Zuck. Probabilistic verification. *Inf. Comput.*, 103(1):1–29, 1993.
29. M. Rabin. The choice coordination problem. *Acta Informatica*, 17:121–134, 1982.
30. W. Shakespeare. *The Tragedy of Macbeth*. ca. 1606. (4.3.225).
31. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS 1986*, pages 332–344, 1986.
32. L. D. Zuck, A. Pnueli, and Y. Kesten. Automatic verification of probabilistic free choice. In *VMCAI 2002,*, volume 2294 of *LNCS*, 2002.

Co-Design and Verification of an Available File System

Mahsa Najafzadeh[1], Marc Shapiro[2], and Patrick Eugster[1,3]

[1] Purdue University, West Lafayette, USA
[2] INRIA-LIP6, Paris, France
[3] Darmstadt University, Darmstadt, Germany

Abstract. Distributed file systems play a vital role in large-scale enterprise services. However, the designer of a distributed file system faces a vexing choice between strong consistency and asynchronous replication. The former supports a standard sequential model by synchronising operations, but is slow and fragile. The latter is highly available and responsive, but exposes users to concurrency anomalies. In this paper, we describe a rigorous and general approach to navigating this trade-off by leveraging static verification tools that allow to verify different file system designs. We show that common file system operations can run concurrently without synchronisation, while still retaining a semantics reasonably similar to Posix hierarchical structure. The one exception is the move operation, for which we prove that, unless synchronised, it will have an anomalous behaviour.

1 Introduction

The market for distributed storage is fueled by cloud computing, big data, exascale computing and so on. Classical file system designs continue to represent approximately 30% of distributed storage needs according to IDC [30], especially for entreprise applications. Modern distributed file system design can improve performance and be highly available by replicating data at several servers. A user can access a file as long as at least one replica of it is available [21, 39]. Ideally, we would like the replicated file system to provide the standard Posix semantics [2], as if a single centralised server handled all operations. However, Posix was designed under the assumption of strong consistency, which requires synchronisation in the critical path. For instance, an operation accessing some file might lock all the directories along the path to the file. This *synchronous*, strong-consistency approach is used in systems such as Frangipani, GFS, GPFS, and Lustre [19, 40, 41, 46]. Although safe, it is unlikely to perform well at large scale, and is unavailable in case of network partition [15].

Experience with real-world applications shows that concurrent updates to the same file system objects are rare [7, 26, 33, 47, 49]. Thus, the synchronous approach often causes more synchronization than the application really needs. Therefore, many recent systems, such as HDFS [43], NFS [34, 38], or PVFS

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 358–381, 2018.
https://doi.org/10.1007/978-3-319-73721-8_17

[12] eschew Posix semantics to gain better performance and scalability. A client reads or writes its local replica without synchronisation with other replicas, and immediately returns to the client, while any updates are transmitted to other replicas in the background. We call this approach *asynchronous replication.*

Unfortunately, asynchronous replication faces the challenge of replica divergence, and may violate some application desirable properties, called *integrity invariants* [35, 48]. Consider this simple example: Alice in Anchorage adds file f to directory d to her local replica, while at the same time, Bob in Brussels removes directory d. The outcome may well result in f existing but being unreachable. Such anomalies are undesirable, and this poses a major challenge to the design of a replicated file system [27].

The correctness of applications implemented on top of a given file system is highly dependent on subtle behaviors of the underlying file system. Thus, programmers require to reason about the file system behaviour, taking into account which anomalies are disallowed by a given file system semantics and whether disallowing these anomalies is enough to ensure correctness.

We address this problem by considering the integrity invariants that both sequential and replicated file systems must satisfy. Our goal is to come up with a design that is *as asynchronous as possible while satisfying the invariants*. We are helped in this road by a static analysis tool based on the CISE logic, which was proved sound for replicated data under the causal consistency assumption [20]. CISE is a variant of rely-guarantee reasoning [25]. A successful analysis proves that any execution of operations over replicated data, under a given synchronisation protocol, maintains the given invariants.

We apply this analysis to file system design. For simplicity, we focus only on a single naming tree and ignore hard links, devices, mounts, file attributes, and the like; we are reasonably confident that our analysis extends readily to a more detailed file system model. The targeted invariant is that the directory structure forms a *tree*. Our model covers the Posix commands [2] affecting the tree structure, including creating, removing, moving, and changing directory entries, as well modifying files.

We first formulate a sequential specification of Posix file system, and prove that this semantics maintains the tree invariant. Next, we extend the sequential semantics to support concurrent users; we study two different concurrent semantics, each exposing a different amount of parallelism, and different anomalies.

The first one, called *Fully Asynchronous*, optimistically accepts all concurrent updates, and resolves conflicts by weakening the sequential Posix semantics. This achieves better availability and latency, which are essential for large-scale applications. Applying the CISE analysis verifies that most operations of a replicated file system can execute without synchronisation. The only exception is that concurrent `move` operations may violate the tree invariant, resulting in a disconnected cyclic component. To fix this issue, our design follows the geoFS system [45]: if a cycle would occur, it effectively replaces `move` with a copy-delete, which preserves the tree invariant but might duplicate the directories that would otherwise end up in a cycle.

If replacing move with copy-delete is undesirable, an alternative safe solution is to use synchronisation in order to disallow the concurrent execution of move operations that would violate the tree invariant. However, several concurrency control algorithms are possible; it is not obvious which is best. Synchronising too much hurts performance and availability, but synchronising too little would result in violating the tree invariant. Using the CISE analysis, we identify the *minimal* synchronization sufficient for move operations. Accordingly, we design our *Mostly-Asynchronous file system*, in which the common operations run in asynchronous mode, and only some move-directory operations might be blocked by synchronisation.

In summary, this paper makes the following contributions:

- We provide a rigorous specification for Posix-like file system for both centralized/synchronous and replicated/asynchronous semantics.
- Using the CISE analysis, we describe a rigorous and general approach that helps developers to encode and verify a variety of concurrent file system designs.
- We study and verify two different replicated file system semantics, each exposing a different amount of parallelism, and different anomalies.
- We identify and verify synchronization is necessary and sufficient for a replicated file system to maintain the tree invariant.

The remainder of the paper is organized as follows: Section 2 introduces the file system objects and models, and gives an overview of the CISE logic. Section 3 presents and verifies a sequential specification of the file system. Section 4 does the same with a concurrent specification of the file system. Sections 5 and 6 discuss our proposed Fully-Asynchronous and Mostly-Asynchronous file system semantics respectively. Related work is discussed in Section 8. Finally, Section 9 summarizes our results and concludes the paper.

2 Model

2.1 File System Objects

The abstract state of a file system consists of a naming tree of *directories*. A directory $d \in \mathsf{Dir}$ maps a set of locally-unique names $n \in \mathsf{Name}$, to a set of *inodes* $\in \mathsf{INode}$. An inode represents a file system object, which is either a directory or a file:

$$\mathsf{Dir} : \mathsf{Name} \rightarrow \mathsf{INode}$$
$$\mathsf{INode} : \mathsf{File}|\mathsf{Dir}$$

This hierarchical file system structure has a single fixed *root* directory. Each inode is identified by a *path*. The path is a sequence of directory names, and possibly a final file name, separated by a separator or delimiter. Following the Unix convention, we use the "/" character as the separator.

We use Greek letters to denote paths.

Definition 1 (Parent Relation). *Directory* $u \in \mathsf{Dir}$ *is the direct parent of inode* $v \in \mathsf{INode}$*, denoted by* $u \downarrow v$*, if and only if* u *contains a mapping to* v*, i.e., there is a name* $n \in \mathsf{Name}$*, such that* $(n, v) \in u$*.*

Definition 2 (Path Prefix). *Path* π *is called a* prefix *of path* γ*, noted* $\pi \sqsubseteq \gamma$*, if and only if* $\gamma = \pi/\alpha$ *for some path* α*.*

The *transitive* closure of the parent relation defines the *ancestor relation.* We say directory $u \in \mathsf{Dir}$ is an ancestor of inode $v \in \mathsf{INode}$, noted $u \downarrow^* v$, if and only if:

$$u \downarrow^* v = \begin{cases} true & if (u \downarrow v) \\ \exists w \in \mathsf{Dir}, (u \downarrow w) \wedge (w \downarrow^* v) & otherwise \end{cases}$$

Definition 3 (Least Common Ancestor). *The Least Common Ancestor of nodes* u *and* v*, noted* $LCA(u, v)$*, is the ancestor of both* u *and* v *that is the farthest from the root. If* u *and* v *are inodes from the same file system then* $LCA(u, v)$ *exists and is unique.*

2.2 Operation Execution Model

We assume a replicated model where each replica stores a full copy of the file system. We use a Read-One-Write-All approach [9], under the operation execution model proposed by Gotsman et al. [20]. A client interacts with the file system through a set of operations Op; it submits an operation to an arbitrary single replica, called the *origin* replica for that operation. The operation is divided into two phases: *generator* and *effector*. The generator executes at the origin replica. This returns a value Val to the client and computes the *effector* of the operation, a function encoding the update to be done by the operation. Every replica eventually applies the effector to its own state.

More precisely, the semantics of operations is defined by a function

$$\mathcal{F} \in \mathsf{Op} \rightarrow (\mathsf{State} \rightarrow (\mathsf{Val} \times (\mathsf{State} \rightarrow \mathsf{State}) \times \mathsf{set}(\mathsf{Token}))).$$

The function is formulated as follows for some operation o:

$$\begin{gathered} \forall \sigma \in \mathsf{State}, o \in \mathsf{Op},\ \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\mathsf{val}}(\sigma), \mathcal{F}_o^{\mathit{eff}}(\sigma), \mathcal{F}_o^{\mathsf{tok}}(\sigma)), \\ \mathcal{F}_o^{\mathsf{val}}(\sigma) \in \mathsf{Val} \\ \mathcal{F}_o^{\mathit{eff}}(\sigma) \in \mathsf{State} \rightarrow \mathsf{State} \\ \mathcal{F}_o^{\mathsf{tok}}(\sigma) \in \mathsf{set}(\mathsf{Token}) \end{gathered}$$

Given state $\sigma \in \mathsf{State}$ at the origin replica in which an operation $o \in \mathsf{Op}$ executes,

- The $\mathcal{F}_o()$ function is the *generator*.
- $\mathcal{F}_o^{\mathsf{val}}(\sigma)$ is the *return value* of operation o. We use $\perp$ for operations that return no value.
- $\mathcal{F}_o^{\mathit{eff}}(\sigma)()$ is the *effector* function of operation o. It will be sent to every replica; when received by a replica, the replica applies this transformation to its own state.
- $\mathcal{F}_o^{\mathsf{tok}}(\sigma)$ is the set of concurrency-control *tokens* acquired by operation o. Tokens are described in Section 2.3.

$$\sigma = \mathsf{map}(\mathsf{Name} \mapsto \mathsf{INode}) \qquad n \in \mathsf{Name} \qquad u \in \mathsf{INode}$$

$$\mathcal{F}_{\mathsf{add}(n)}(\sigma) \ : \ \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{add}(n)}(\sigma) = \bot, \mathcal{F}^{\mathsf{tok}}_{\mathsf{add}(n)}(\sigma) = \emptyset, u = \mathsf{inode}() \\ \mathcal{F}^{\mathit{eff}}_{\mathsf{add}(n,u)}(\sigma) = \lambda(\sigma').\, \sigma'[n \mapsto u] \end{cases}$$

$$\mathcal{F}_{\mathsf{remove}(n)}(\sigma) \ : \ \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{remove}(n)}(\sigma) = \bot, \mathcal{F}^{\mathsf{tok}}_{\mathsf{remove}(n)}(\sigma) = \emptyset \\ \mathcal{F}^{\mathit{eff}}_{\mathsf{remove}(n)}(\sigma) = \lambda(\sigma').\, \sigma'[n \mapsto \emptyset] \end{cases}$$

$$\mathcal{F}_{\mathsf{query}(n)}(\sigma) \ : \ \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{query}(n)}(\sigma) = u \mid \sigma[n \mapsto u] \\ \mathcal{F}^{\mathsf{tok}}_{\mathsf{query}(n)}(\sigma) = \emptyset \\ \mathcal{F}^{\mathit{eff}}_{\mathsf{query}(n)}(\sigma) = \mathsf{skip} \end{cases}$$

Precondition	Operation
$\nexists e \in \mathsf{INode}, (n, e) \in \sigma$	$\mathcal{F}_{\mathsf{add}(n)}(\sigma)$
$\exists e \in \mathsf{INode}, (n, e) \in \sigma$	$\mathcal{F}_{\mathsf{remove}(n)}(\sigma)$
$\exists e \in \mathsf{INode}, (n, e) \in \sigma$	$\mathcal{F}_{\mathsf{query}(n)}(\sigma)$

Fig. 1. A sequential specification of directory

2.3 Concurrency

A replica is a process that executes a sequence of generator and effector events. An operation o is *visible* to operation o' if some replica executes the effector of o before the generator of o'. We assume *causal consistency*, i.e., the visibility relation is transitive. Two operations that are not related by visibility are said *concurrent*.

The tokens mentioned in the previous section are an abstraction of concurrency control mechanisms. Tokens are related by a symmetric *conflict* relation ($\bowtie$). If two operations acquire tokens conflicting according to $\bowtie$, then one must be visible to the other. If their tokens do not conflict by $\bowtie$, the operations are allowed to be concurrent.

2.4 Example

Consider a directory d in the file system. Figure 1 illustrates a simple implementation of d as follows: Let σ denote the state of d at the origin replica. State σ is a map of names to inodes, representing the content of the directory. The directory semantics supports operations to add, remove, and query mappings in the directory.

We start with a sequential specification for the directory, shown in Fig. 1. To add an inode to d, the $\mathsf{add}(n)$ operation's generator in the origin replica, computes return value, tokens, and creates a new inode to prepare its effector if σ satisfies the operation's precondition. The precondition of $\mathsf{add}(n)$ is that no inode under the same name n exists in the directory d. We assume a function $u = \mathsf{inode}()$ to create a new and unique inode identifier u. The effector for $\mathsf{add}(n)$ takes name n and inode u as its arguments, reads state of d at the current replica, denoted σ' (which can be different from σ, due to concurrency), and then adds the inode u under name n to it, denoted by $\sigma'[n \mapsto u]$. Similarly, to remove

an inode named n from directory d, the $\mathsf{remove}(n)$ operation's effector, reads the state of directory at each replica, denoted by σ', and removes the mapping for name n, noted by $\sigma'[n \mapsto \emptyset]$. The $\mathsf{query}(n)$ operation computes the inode u mapped to the name n at the origin replica, and simply returns u, i.e., $\sigma[n \mapsto u]$; its effect is simply skip where $\mathsf{skip} = (\lambda\sigma'.\sigma')$.

2.5 CISE Logic and Analysis

Any replicated system needs to ensure convergence, i.e., executing the same operations produces the same results at different replicas. Furthermore, a given system must maintain a specific integrity *invariant*, i.e., a safety condition over replica states. The invariant of interest for file systems is that its directories form a tree (the *tree invariant*). In this work, we do not consider liveness properties.

To this effect, we leverage CISE [20], a sound logic that allows to verify such safety conditions statically (in polynomial time) for a replicated system with causal consistency. We use the CISE verification tool [31], which takes as input the specification of the system's operations (their preconditions and effects) and of the targeted invariant. The tool checks the following proof rules: *(i)* Individual Correctness: each operation individually maintains the invariant. *(ii)* Commutativity: any two concurrent operations commute (operations that have conflicting tokens need not commute). *(iii)* Stability: every operation's precondition is stable under concurrent updates (but not necessarily against an operation that acquires a conflicting token). A successful analysis proves that *any* execution of the given operations, under the given tokens, maintains the given invariant.

Unsuccessful analysis returns a counter-example, which indicates the problem to the developer. The developer can fix the problem, either by weakening the application semantics, or by strengthening the token system, and then run the check again. This process constitutes *co-design* of the application semantics and of its concurrency control.

3 Sequential Specification

In this section, we first formulate a sequential specification of a Posix-like file system, and prove that this semantics preserves the tree invariant.

3.1 Tree Invariant

An directory is a map of names to inodes (files or directories). We model the directory structure as a graph, where a directory is a node, and there is an edge from this *parent* directory to each directory that it names (its *children*). The tree main invariant I of the file system is the conjunction of the following assertions: (1) The file system has a fixed root node. (2) The root is an ancestor of every other node in the tree. (3) Every node has exactly one parent, except the root, which has none. (Since we ignore symbolic links, a file cannot have several parent

$$\sigma = \mathsf{set}(\mathsf{INode}) \quad \sigma_{init} = \{\mathsf{r}oot\} \quad n \in \mathsf{Name} \quad c \in \mathsf{Content}$$
$$d, d_p, d_{p_{new}}, d_{p_{old}} \in Dir \quad f \in \mathsf{File}$$

$$\mathsf{mkdir(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), d = \mathsf{inode}() \\ \mathcal{F}^{eff}_{\mathsf{mkdir}(d_p,n,d)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto d] \end{cases}$$
$$\mathsf{rmdir(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), \\ \mathcal{F}^{eff}_{\mathsf{rmdir}(d_p,n)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto \emptyset] \end{cases}$$
$$\mathsf{mkfile(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), f = \mathsf{inode}() \\ \mathcal{F}^{eff}_{\mathsf{mkfile}(d_p,n,f)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto f] \end{cases}$$
$$\mathsf{rmfile(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), \\ \mathcal{F}^{eff}_{\mathsf{rmfile}(d_p,n)}(\sigma) = \lambda(\sigma').\, \sigma'.d_p[n \mapsto \emptyset] \end{cases}$$
$$\mathsf{write(path, c)} : \begin{cases} f = \mathcal{L}(\mathsf{path}), \\ \mathcal{F}^{eff}_{\mathsf{write}(f,c)}(\sigma) = \lambda(\sigma').\, \sigma'.f.\mathsf{write}(c) \end{cases}$$
$$\mathsf{mvdir(old, new)} : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi), d = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma), \\ \mathcal{F}^{eff}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}(\sigma) = \lambda(\sigma').\, (\sigma'.d_{p_{old}}[n \mapsto \emptyset]; \sigma'.d_{p_{new}}[n \mapsto d]) \end{cases}$$
$$\mathsf{mvfile(old, new)} : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi), f = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma), \\ \mathcal{F}^{eff}_{\mathsf{mvfile}(d_{p_{old}},n,d_{p_{new}},f)}(\sigma) = \lambda(\sigma').\, (\sigma'.d_{p_{old}}[n \mapsto \emptyset]; \sigma'.d_{p_{new}}[n \mapsto f]) \end{cases}$$

Precondition	Operation
$\nexists e \in \mathsf{INode}, (n, e) \in d_p \wedge \mathsf{r}oot \downarrow^* d_p$	$\mathcal{F}_{\mathsf{mkdir}(d_p,n,d)}(\sigma)$
$\nexists e \in \mathsf{INode}, m \in \mathsf{Name}, (m, e) \in d \wedge \mathsf{r}oot \downarrow^* d$	$\mathcal{F}_{\mathsf{rmdir}(d_p,n,d)}(\sigma)$
$\nexists e \in \mathsf{INode}, (n, e) \in d_p \wedge \mathsf{r}oot \downarrow^* d_p$	$\mathcal{F}_{\mathsf{mkfile}(d_p,n,f)}(\sigma)$
$\mathsf{r}oot \downarrow^* f$	$\mathcal{F}_{\mathsf{rmfile}(d_p,n,f)}(\sigma)$
$\mathsf{r}oot \downarrow^* f$	$\mathcal{F}_{\mathsf{write}(f,c)}(\sigma)$
$\nexists e \in \mathsf{INode}, (n, e) \in d_{p_{new}} \wedge \mathsf{r}oot \downarrow^* d \wedge d_{p_{old}} \downarrow d$ $\wedge \mathsf{r}oot \downarrow^* d_{p_{new}} \wedge d \not\downarrow^* d_{p_{new}}$	$\mathcal{F}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}(\sigma)$
$\nexists e \in \mathsf{INode}, (n, e) \in d_{p_{new}} \wedge \mathsf{r}oot \downarrow^* f$ $\wedge d_{p_{old}} \downarrow f \wedge \mathsf{r}oot \downarrow^* d_{p_{new}}$	$\mathcal{F}_{\mathsf{mvfile}(d_{p_{old}},n,d_{p_{new}},f)}(\sigma)$

Fig. 2. A sequential hierarchy file system design

directories.) (4) The directory graph is acyclic. (5) The names in a directory are unique with respect to that directory. Formally:

$$\begin{aligned} I = \forall e_1, e_2 \in \mathsf{INode}, d_1, d_2 \in \mathsf{Dir}, n_1, n_2 \in \mathsf{Name}, \pi, \pi' \in \mathsf{Path} \\ (1) \quad \mathsf{r}oot \in \mathsf{INode} \\ (2) \wedge e_1 \neq \mathsf{r}oot \implies \mathsf{r}oot \downarrow^* e_1 \\ (3) \wedge (d_1 \downarrow e_1 \wedge d_2 \downarrow e_1 \implies d_1 = d_2 \wedge e_1 \neq \mathsf{r}oot) \\ (4) \wedge (d_1 \downarrow^* d_2 \wedge d_2 \downarrow^* d_1 \implies d_1 = d_2) \\ (5) \wedge (\pi/n_1 \mapsto e_1 \wedge \pi/n_2 \mapsto e_2 \wedge e_1 \neq e_2 \implies n_1 \neq n_2) \end{aligned}$$

3.2 File System Operations

The file system semantics that we study in this paper consists of a set of operations, which abstract the Posix commands to manipulate the tree structure and

to update file content. They include *creating*, *deleting*, and *renaming* directories or files, and modifying files. We identify a file or a directory by a path. Given the path argument of an operation, a *resolution* function $\mathcal{L}$ is executed at the operation's origin replica to find the inode located in the path,

$$\mathcal{L} : \mathsf{Path} \rightarrow \mathsf{INode}.$$

Figure 2 shows a sequential specification of the file system. We denote σ the state of the file system, and the dot notation, for instance $\sigma.e$, to refer to a particular inode e in it.

We assume that update operations return no value, i.e., $\mathcal{F}_o^{\mathsf{val}}(\sigma) = \perp$, and an exclusive token is assigned to each operation that forbids the concurrent execution of these operations. For complex operations like move operation, whose effect updates two directories, indicated by old and new, we use semicolon to denote a sequence of changes over the file system state. We assume that inodes have unique identifiers across replicas. The arguments to the effector of some operation includes the inode determined by the generator, rather than its path; this is unnecessary in the sequential specification but will prove useful when we consider concurrent updates.

For instance, consider that Alice wants to create a new directory using the operation mkdir(*/share/album/paris*). In Alice's replica of the file system, this resolves to creating name to name *paris*, within the directory whose path is */share/album*, which evaluates to inode d_p (subscript p stands for "parent"). A minimal precondition is that d_p exists, and it does not contain name *paris*. If satisfied, effector $\mathcal{F}^{eff}_{\mathsf{mkdir}(d_p,"paris",d)}$ is generated and sent to all replicas. On delivering the effector, every replica (including Alice's) applies its effect, which is to create the new directory d, and to update the parent directory d_p by mapping the name "*paris*" to directory d, denoted by $d_p[paris \mapsto d]$. Note, we use the notation f.write(c) for writing the content c into a file f.

We apply the CISE effector safety to check if the file system operations preserve the tree invariant in isolation, with the minimal preconditions of the previous paragraph (since no operations are concurrent, the other two rules are void). The CISE tool returns an error for mvdir. The associated counter-example shows that the source directory must not be an ancestor of the destination directory, because otherwise, a cycle occurs. The developer strengthens the pre-condition to avoid this; with this stronger specification, the tool indicates that mvdir is now safe. We proceed similarly for the other operations, thus co-designing the sequential specification. Finally, we reach the specification illustrated in Fig. 2, which is proved safe.

4 Concurrent Specification

In this section, we extend the sequential semantics to support concurrent users. We present a concurrent specification of the file system that *optimistically* accepts all concurrent updates, and resolves conflicts but it trades the sequential

$$\sigma = (d \times \rho) \quad d = \mathsf{map}(\mathsf{Name} \mapsto \mathsf{INode}) \quad \rho = \mathsf{map}(\mathsf{INode} \mapsto \mathsf{INode})$$
$$n \in \mathsf{Name} \quad u, v, w, e_1, \ldots, e_k \in \mathsf{Dir}$$
$$\mathcal{F}_{\mathsf{add}(n)}(\sigma) : \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{add}(n)}(\sigma) = \bot, \quad \mathcal{F}^{\mathsf{tok}}_{\mathsf{add}(n)}(\sigma) = \emptyset, u = \mathsf{inode}() \\ \mathcal{F}^{\mathit{eff}*}_{\mathsf{add}(n,u)}(\sigma) = \lambda(\sigma').\,(\mathsf{IF}(\sigma'.d[n \mapsto v] \wedge w == v \bigoplus u) \\ \qquad (\sigma'.d[n \mapsto w], \sigma'.\rho[u \mapsto w, v \mapsto w])(\sigma'.d[n \mapsto u], \sigma'.\rho) \end{cases}$$
$$\mathcal{F}_{\mathsf{remove}(n)}(\sigma) : \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{remove}(n)}(\sigma) = \bot, \quad \mathcal{F}^{\mathsf{tok}}_{\mathsf{remove}(n)}(\sigma) = \emptyset, \\ \mathcal{F}^{\mathit{eff}*}_{\mathsf{remove}(n,u)}(\sigma) = \lambda(\sigma').\,(\mathsf{IF}(\sigma'.d[n \mapsto w] \wedge \sigma'.\rho[u \mapsto w]) \\ \qquad (\sigma'.d, \sigma'.\rho[u \mapsto \emptyset])(\sigma'.d[n \mapsto \emptyset], \sigma'.\rho) \end{cases}$$
$$\mathcal{F}_{\mathsf{query}(n)}(\sigma) : \begin{cases} \mathcal{F}^{\mathsf{val}}_{\mathsf{query}(n)}(\sigma) = w \mid d[n \mapsto w] \wedge (\rho[e_1 \mapsto w, e_2 \mapsto w, \ldots, e_k \mapsto w] \\ \qquad \implies w = e_1 \bigoplus e_2 \ldots \bigoplus e_k) \\ \mathcal{F}^{\mathsf{tok}}_{\mathsf{query}(n)}(\sigma) = \emptyset, \quad \mathcal{F}^{\mathit{eff}*}_{\mathsf{query}(n)}(\sigma) = \mathsf{skip} \end{cases}$$

Fig. 3. A concurrent specification of directory d

Posix semantics for availability. The concurrent design exploits Conflict-Free Replicated Data Types (CRDT) [42] to address conflicts. CRDTs include many useful data types, such as counters, sets, graphs, and maps, which encapsulate conflict resolution policies for automatically merging the effects of operations performed on each object concurrently.

4.1 File System Objects as CRDTs

We use the idea of CRDTs to carefully design conflict-free replicated file system objects, which ensures convergent outcomes reflecting effects of all operations performed to each file or directory at different replicas. To this goal, we first discuss conflict cases that may occur as a result of concurrent execution of file system operations, and then propose the concurrent file system semantics, which converges by design.

4.2 Name Conflict

Users may perform concurrent updates to a directory. Concurrently adding or moving inodes under the same name in the same directory is problematic because it violates the name uniqueness property in the tree invariant (*name conflict*).

To address such conflicts, we define a merge operator $\bigoplus$. The merge operator $\bigoplus$ may have different merge semantics depending on directories or files, or may be different for different files. We choose the following merge semantics: concurrently adding or moving two directories under the same name to the same parent directory merges these directories, i.e., takes their union. For files with the same name, the merge semantics renames files by appending a replica-specific suffix to a locally-unique file name. We assume that type of each file system object is embedded in its name, and hence name conflicts between files and directories cannot occur.

Definition 4 (Union Merge Function). *Let u and v be two different directories with the same name n under parent directory d_p. We define the union merge of u and v as follows:*

$$w = u \bigoplus v \mid d_p[n \mapsto w] \ \wedge (\forall e \in \mathsf{INode}, u[- \mapsto e] \vee v[- \mapsto e] \implies w[- \mapsto e])$$

Where w is a new directory whose content is union of contents of directories u and v. The merge function is recursively applied to sub-directories if there are naming conflicts.

A concurrent effector may still use the old directories u and v. To solve this problem, each replica's state has map ρ, which keeps a record of the equivalence relation between the directories and their merged directory e.g., $\rho[u \mapsto w, v \mapsto w]$ where $w = u \bigoplus v$. Thus, when a replica receives an effector updating a directory, the replica first queries the equivalence relation ρ to check if the directory has been merged. Unused identifiers can be garbage-collected and removed from ρ. For brevity, we do not attempt to formalise this property.

Definition 5 (Rename Merge Function).
Let u and v be two files with the same name n under the same parent directory d_p, which originally are generated in replica r_1 and r_2, respectively. A merge decision to solve the file name conflicts would be to change the names by adding the replica's suffixes to the original name. Thus, we define the rename merge function of files u and v as follows:

$$u \bigoplus v : d_p[n_1 \mapsto u, n_2 \mapsto v]$$

Where $n_1 = n + r_1$ and $n_2 = n + r_2$ are new unique names mapped to u, and v respectively.

The merge operator must satisfy the following properties:

- $u \bigoplus u = u \quad (idempotent)$
- $u \bigoplus v = v \bigoplus u \quad (commutative)$
- $u \bigoplus (v \bigoplus w) = (u \bigoplus v) \bigoplus w \quad (assosiative)$

where u, v, and w are file system objects.

Figure 3 illustrates the concurrent implementation of directory d using the merge operator. $\mathcal{F}_o^{eff^*}$ is the effector of operation o that integrates the merge policy for managing name conflicts. The directory's state σ consists of two maps: d that is a map of names to inodes, representing the directory's content, and ρ that is a map of inodes to inodes, tracking the equivalence relation of merged sub-directories in d. The $\mathsf{add}(n)$ operation creates a new inode u; its effector reads the directory's state at each replica, denoted by σ', and if there is no name conflict, it simply adds the pair (n, u) to the directory's content, denoted by $\sigma'.d[n \mapsto u]$. For simplicity, we assume the name n refers to a directory. In the case where a directory v is concurrently added under the same name n,

$$\sigma = \mathsf{set(INode)} \quad \sigma_{init} = \{root\} \quad \mathsf{Token} = \emptyset \quad \bowtie = \emptyset$$

$$\mathsf{mkdir(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), d = \mathsf{inode()} \\ \mathcal{F}^{eff}_{\mathsf{mkdir}(d_p,n,d)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(\sigma'.d_p); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{rmdir(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), d = \mathcal{L}(\mathsf{path}) \\ \mathcal{F}^{eff}_{\mathsf{rmdir}(d_p,n,d)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{mkfile(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), f = \mathsf{inode()} \\ \mathcal{F}^{eff}_{\mathsf{mkfile}(d_p,n,f)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{add}(n,f)}(\sigma'.d_p); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{rmfile(path)} : \begin{cases} \mathsf{path} = \pi/n, d_p = \mathcal{L}(\pi), f = \mathcal{L}(\mathsf{path}) \\ \mathcal{F}^{eff}_{\mathsf{rmfile}(d_p,n,f)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{write(path, c)} : \begin{cases} f = \mathcal{L}(\mathsf{path}) \\ \mathcal{F}^{eff}_{\mathsf{write}(f,c)}(\sigma) = \lambda(\sigma').\,\sigma'.f.\mathsf{write}(c); \mathsf{recurAdd}(\sigma'.f) \end{cases}$$

$$\mathsf{mvdir(old, new)} : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi) \wedge d = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma) \\ \mathcal{F}^{eff}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}(\sigma) = \lambda(\sigma').\,(\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_{p_{old}}); \\ \qquad \mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(\sigma'.d_{p_{new}}); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

$$\mathsf{mvfile}(old, \mathsf{new}) : \begin{cases} \mathsf{old} = \pi/n, \mathsf{new} = \gamma/n, d_{p_{old}} = \mathcal{L}(\pi) \wedge f = \mathcal{L}(\mathsf{old}), d_{p_{new}} = \mathcal{L}(\gamma) \\ \mathcal{F}^{eff}_{\mathsf{mvfile}(d_{p_{old}},n,d_{p_{new}},f)}(\sigma) = \lambda(\sigma').\,((\mathcal{F}^{eff^*}_{\mathsf{remove}(n)}(\sigma'.d_{p_{old}}); \\ \qquad \mathcal{F}^{eff^*}_{\mathsf{add}(n,f)}(\sigma'.d_{p_{new}}); \mathsf{recurAdd}(\sigma'.d_p)) \end{cases}$$

Fig. 4. A concurrent specification of the file system (fully asynchronous)

in such a way that $w = u \bigoplus v$, the effector adds the pair (n, w) to the directory's content, and updates the directory's equivalence relation ρ by adding the corresponding equivalence into, denoted by $\sigma'.\rho[u \mapsto w, v \mapsto w]$. Similarly, the $\mathsf{remove}(n)$ operation's effector simply removes the mapping for name n from the directory's content at each replica, denoted by $\sigma'.d[n \mapsto \emptyset]$. If u is a directory which is merged into new directory w due to name conflicts, i.e., $\sigma'.d[n \mapsto w]$ and $\sigma'.\rho[u \mapsto w]$, the effector removes equivalence mapping for u from the relation ρ at each replica, i.e., $\sigma'.\rho[u \mapsto \emptyset]$. The $\mathsf{query}(n)$ operation computes the directory mapped to the name n by first reading the relation ρ at the origin replica and computing all directories $e_{i_{1 \le i \le k}}$ concurrently added to the directory's content d under the name n, and returns w, where $w = e_1 \bigoplus e_2 \bigoplus \ldots \bigoplus e_k$.

4.3 Remove/Update Conflict

A different kind of conflict happens when a replica updates an inode, while another replica concurrently removes the inode. This kind of conflict is called a *remove/update conflict*. For instance, when a replica receives an operation to add directory u to directory v, if directory v has been removed by a concurrent user, the operation execution results in an unreachable directory u.

The replicated data types support two main approaches, called *add-wins* and *remove-wins*, to address this problem. They differ by the result of concurrent

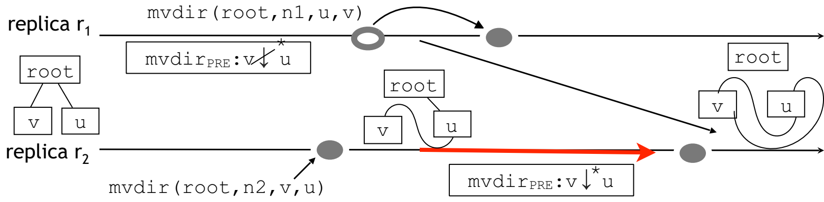


Fig. 5. Counter-example for violation of tree invariant due to of concurrent moves

add and remove of the same elements. In the add-wins semantics, when there are concurrent add or remove of the same element, add wins and the effects of concurrent remove operations are ignored. Remove-wins follows the opposite semantics. When an element is removed, any concurrent updates of the same element are lost.

Given the directory semantics in Fig. 3, concurrent adding inodes into the same directory d commute since each inode is unique. Concurrent removes commute because removing different inodes has independent effects, and removing the same inode is identical. Moreover, concurrent adding and removing inodes to the same directory d commute, i.e., the add wins because the unique inode created by add operation cannot be observed by remove operation.

Figure 4 illustrates a concurrent specification of the file system. The concurrent semantics is token-free, and relies on the effectors $\mathcal{F}^{eff^*}_{\mathsf{add}}$, and $\mathcal{F}^{eff^*}_{\mathsf{remove}}$ presented in Fig. 3 to handle name conflicts occurring within a directory. Following the add-wins semantics, function $\mathsf{recurAdd}(d)$ recursively re-creates the removed directories, which have been concurrently updated. The function takes an inode, e.g., directory d, as input, and then checks whether the directory is reachable by the root, if not, the *full* directory's path from the root is re-created as follows:

$$\mathsf{recurAdd}(d) = \mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(d.\mathsf{Parent}) + \mathsf{recurAdd}(d.\mathsf{Parent})$$

where $d.\mathsf{Parent}$ is parent of d. The function $\mathsf{recurAdd}(d)$ uses the effector $\mathcal{F}^{eff^*}_{\mathsf{add}(n,d)}(d.\mathsf{Parent})$ to re-add a removed directory d into its parent directory.

For instance, consider the concurrent semantics of $\mathsf{mkdir}(\mathsf{path})$ operation for creating a new empty directory identified by the path argument. In the origin replica, this operation resolves to creating name to name n, within the directory whose path is π, which evaluates to inode d_p. Its sequential semantics is to create the new directory d, and to update the parent directory d_p by mapping the name n to directory d. However, concurrent conflicts may arise: 1) other directories concurrently added or moved into parent directory d_p under the same name n, 2) the directory d_p has been removed concurrently. To address name conflicts when applying mkdir effector on directory d_p in any replica state σ', the concurrent semantics uses the name conflict resolutions given by $\mathcal{F}^{eff^*}_{\mathsf{add}(n,f)}(\sigma'.d_p)$.

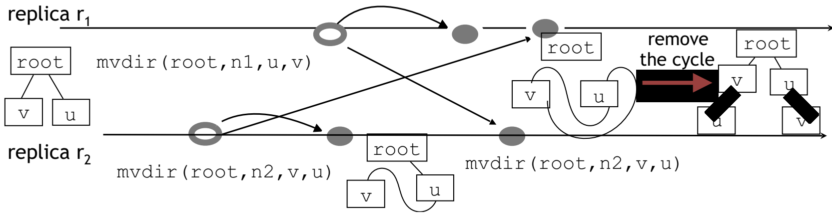


Fig. 6. Asynchronous solution design for conflicting move operations

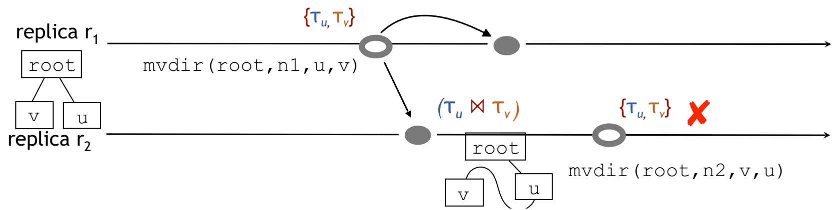


Fig. 7. Avoiding concurrent execution of conflicting moves

The concurrent semantics also uses function recurAdd$(\sigma'.d_p)$, which re-creates directory d_p if it has been removed concurrently from the replica state σ'.

We remark that removing a directory does not actually delete the directory, as it only removes the mapping for the directory from its parent, as the directory exists but is unreachable. Re-creating a directory adds it into its parent directory again. Therefore, if multiple replicas concurrently re-create directory d, they all will end up with the same state.

We use the CISE-enabled analysis to verify the concurrent design. The commutativity analysis verifies that the concurrent operations results in a convergent state because all possible pairs of concurrent operations commute. The application of the CISE stability analysis for the concurrent file system design verifies that most operations of the file system can execute without synchronisation, and only concurrent move operations may violate the tree invariant. The precondition of move directory operation is not *stable* when there is another concurrent move operation. Figure 5 illustrates a counter-example: consider a file system with three directories, $root$, u and v, replicated at two replicas. Initially, the $root$ is parent of u and v. One replica asks to move directory u named n under directory v using the move operation mvdir$(root, n, u, v)$. The precondition of this move operation is true, i.e., the directory u is not an ancestor of directory v. However, concurrently, another replica moves directory v under directory u, and hence, the precondition of move is not true any more. If indeed we were to continue and apply the effect of the first move operation, we come to the state, with a cycle of u and v, disconnected from the $root$. Obviously, it is not a tree.

Table 1. Required tokens for the mostly-asynchronous file system

Effector	Tokens
$\mathsf{mvdir}(d_{p_{old}}, n, d_{p_{new}}, d)$	$\{\ \tau_d, \tau_{d_{p_{new}}}\} \cup$ $\{\tau_e \mid e \in \mathsf{Dir} \wedge e \downarrow^* d_{p_{new}} \wedge$ $LCA(d, d_{p_{new}}) \downarrow^* e\}$
$\mathsf{mvfile}(d_{p_{old}}, n, d_{p_{new}}, f)$	τ_f

5 Fully-Asynchronous File System

If high performance and availability of update operations are important to the file system application, a simple approach to fix move conflicts is to allow move operations to execute without restriction, and to repair the tree invariant violations after the fact. Thus, we design a fully-asynchronous file system that accepts all concurrent operations, and if cycles occur due to concurrently moving two directories u, and v, it has a merge function, which will duplicate all directories in the cycle. For instance, in Fig. 6, the cycle between directories u and v is removed by making copies of the directories u and v. This merge semantics is used in real file systems, such as geoFS [45].

We use the CISE analysis to verify the merge semantics for move operations. The analyser proves that the merge function is commutative, and also guarantees the tree invariants.

6 Mostly-Asynchronous File System

One alternative approach for handling move conflicts is to add synchronisation in order to avoid concurrent execution of move operations that would violate the tree invariant. Thus, we co-design the file system semantics, in which the common operations run in asynchronous mode, and only some move directory operations need to be synchronised.

A developer may define a mutually exclusive token for each inode $e \in \mathsf{INode}$: τ_e, such that $\tau_e \bowtie \tau_e$. To ensure that cycles do not happen, we assign a set of tokens to each move operation. For any pair of move operations, if their tokens are conflicting, only one of them can take effect, because token semantics requires that the operations exchange messages, which ensures that one of the operations is aware of the other. However, other move operations are causally independent, and hence can proceed in parallel.

Using the CISE analysis, we identify and verify the necessary and sufficient token assignments for move directory operations ensuring the tree invariant in any possible executions.

Lemma 1. *Assume a* move *directory operation moving directory d into its destination directory $d_{p_{new}}$. Let set A be the set of ancestors of the destination directory $d_{p_{new}}$ up to $LCA(d, d_{p_{new}})$. Token set $T = \{\tau_d, \tau_{d_{p_{new}}}\} \cup \{\tau_e \mid e \in A\}$ represents the necessary and sufficient tokens required by the* move *directory operation.*

Proof. First, we follow the CISE analysis to prove that the set T is indeed sufficient for maintaining the tree invariant. To this goal, we prove that precondition of move operation is stable against all concurrent move operations allowed by the tokens. Assume operation $\mathcal{F}_{\mathsf{mvdir}(d_{p_{old}},n,d_{p_{new}},d)}$ moving directory d located in the parent directory $d_{p_{old}}$ to its new parent directory $d_{p_{new}}$. The operation is associated with tokens τ_d, and $\tau_{d_{p_{new}}}$ over the directory d and $d_{p_{new}}$, and a set of tokens τ_e, for all directories e, which are ancestors of the new parent directory $d_{p_{new}}$ up to $LCA(d, d_{p_{new}})$.

The precondition of move operation requires that (1) directory $d_{p_{old}}$ is the parent of directory d, and (2) directory d is not reachable from $d_{p_{new}}$. The first assertion becomes false when another move operation concurrently moves directory d. Token τ_d in set T disallows this concurrent execution. The second assertion will be violated when another move operation concurrently move directory $d_{p_{new}}$ under directory d. Acquiring token $\tau_{d_{p_{new}}}$ and the tokens over $d_{p_{new}}$'s ancestors in set T forbids such concurrent situation.

However, we only need to acquire tokens over ancestors of $\tau_{d_{p_{new}}}$ up to the least common ancestor of d and $\tau_{d_{p_{new}}}$. We use contradiction to support this claim. For brevity, we use $\mathsf{mvdir}(d_{p_{new}}, d)$ to indicate moving directory d in to directory $d_{p_{new}}$. Let assume that the ancestors' tokens up to the $LCA(d, d_{p_{new}})$ is not sufficient, and a cycle is created as follows:

$$a \downarrow c \ldots \downarrow d_{p_{new}} \downarrow d \ldots b \downarrow a$$

Where c, b, a are directories. This happens when there are move operations concurrently moving the $d_{p_{new}}$'s ancestors, which are located above the $LCA(d, d_{p_{new}})$. The left side of this cycle ($a \downarrow c \ldots \downarrow d_{p_{new}} \downarrow d$) indicates that there is an operation $\mathsf{mvdir}(a, c)$ concurrently moving one of common ancestors of $d_{p_{new}}$ and d, say c, in to directory a. This operation succeeds iff directory a is not a descendant of c (it's the move's precondition).

Now, consider the right side ($b \downarrow a$), where another concurrent operation $\mathsf{mvdir}(b, a)$ moves directory a in to one of d's descendants, say b. This operation requires tokens over directory b up to $LCA(b, a)$. Depending on the location of $LCA(b, a)$, we consider two cases: 1) directory d is located between $LCA(b, a)$ and the destination directory b, i.e., directory d is in set A of $\mathsf{mvdir}(b, a)$ operation ($LCA(b, a) \downarrow^+ d \downarrow^+ b$). Thus, moving a to b requires token over d, which conflicts with the token set of $\mathsf{mvdir}(d_{p_{new}}, d)$ operation. 2) $LCA(b, a)$ is located under d. This means that directory a is d's descendant. Knowing c is d's ancestor, a is also c's descendant. This violates the precondition of operation $\mathsf{mvdir}(a, c)$, which requires directory c not to be a's ancestor, and hence, the execution of operation $\mathsf{mvdir}(a, c)$ cannot happen. Unless, there was another operation $\mathsf{mvdir}(d, a)$ moving directory a in to directory d concurrently with operation $\mathsf{mvdir}(a, c)$. However, this move operation also requires token d conflicting with the tokens of $\mathsf{mvdir}(d_{p_{new}}, d)$ operation.

Thus, independent of LCA's location, the right and left hand side of cycle cannot be true at the same time, i.e., directory a cannot move in to one of d's descendants while moving d's ancestors in to a. This contradicts the original

assumption that the cycle is created, and the above is impossible. Therefore, we conclude that acquiring tokens up to $LCA(d, d_{p_{new}})$ is sufficient.

Now, we show that set T is necessary, i.e., it contains the minimal set of tokens, by contradiction: We assume that T is not minimal, meaning that it includes unnecessary tokens. We remove a token $\tau \in T$, and then check whether concurrent executions of move operations still maintain the tree invariant. If so, set T is not minimal.

1. τ_d is the token over the source directory d. Removing token τ_d from set T allows concurrent operations to move the same directory d to another destination directory d_1. If $d_1 \neq d_{p_{new}}$, then d will have two parents; violating the tree invariant.
$$d_{p_{new}} \downarrow d \wedge d_1 \downarrow d$$
2. $\tau_{d_{p_{new}}}$ is the token over destination directory $d_{p_{new}}$. Removing token $\tau_{d_{p_{new}}}$ from set T allows another move operation to concurrently move destination directory $d_{p_{new}}$ to directory d_1. If $d_1 = d$, or if directory d_1 is a descendent of directory d, i.e., $d \downarrow^+ d_1$, then cycles occur.
$$d_1 \downarrow d_{p_{new}} \wedge d_{p_{new}} \downarrow d \wedge d \downarrow^+ d_1$$
3. τ_{d_1} is the token over one of $d_{p_{new}}$'s ancestors, say d_1. . Removing token τ_{d_1} from set T allows another move operation to concurrently move directory d_1 to directory d_2. If $d_2 = d$, or if directory d_2 is a descendent of source directory d, i.e., $d \downarrow^+ d_2$, then cycles occur.
$$d_2 \downarrow d_1 \wedge d_1 \downarrow^+ d_{p_{new}} \wedge d_{p_{new}} \downarrow d \wedge d \downarrow^+ d_2$$

Thus, for moving directory d, we only require to acquire tokens over d, its destination directory $\tau_{d_{p_{new}}}$, and all ancestors of $\tau_{d_{p_{new}}}$ up to $LCA(d, d_{p_{new}})$. Concurrent move operations are allowed as long as their token sets are compatible. The intuition behind acquiring tokens over ancestors up to the least common ancestor is: if a directory is a common ancestor of directory d and its destination directory, the directory cannot be involved in concurrent move operations that result into the tree invariant's violation because it is disallowed by the move operation's precondition.

For instance, Fig. 7 illustrates how the token assignment avoids previous counter example. Operation mvdir($root, n_1, v, u$) acquires tokens $\{\tau_u, \tau_v\}$, and operation mvdir($root, n_2, u, v$) acquires the tokens $\{\tau_u, \tau_v\}$. Their token sets are not compatible, token τ_u is not compatible with token τ_v, i.e., $\bowtie = \{(\tau_u, \tau_v)\}$. When another move operation executes at a different replica r_2. This will force it to synchronise with other replicas to find out if there are other move operations. So, it will get the information about the first move operation in replica r_1, and cannot succeed.

We add the corresponding tokens to the move semantics, and perform the CISE stability analysis again. This time, the tool generates a counter-example

Table 2. A summary of file semantics verified by tool

Semantics	#Op	#Tokens	#Invariant	Anomaly	Verification Time(ms)
Sequential Design	7	7	1	NO	278
Concurrent Design	7	0	1	invariant violation	1297
Fully-Asynchronous Design	7	0	1	duplication	2350
Mostly-Asynchronous Design	7	2	1	NO	1570

that indicates that two concurrent users might move the same file to different locations. Thus, the file would end up with two parent directories; violating the tree invariant. To avoid this issue, we assign an exclusive move token τ_f over file f to each move file operation. Table 1 presents the move tokens required in the mostly- asynchronous file system design. The semantics successfully passes all three CISE analyses. The analyser proves that the consistency choices for different move operations are sufficient to preserve the tree invariant.

7 Evaluation

We have developed a verification tool that leverages the CISE analysis to co-design and verify a replicated file system. Our tool is currently implemented as a few hundred lines of Java code that reduces the CISE obligations to Satisfiability Modulo Theories (SMT) queries. We built the tool on the Z3 SMT solver [1], developed by Microsoft Research for the verification and analysis of software applications. Using the tool, we are able to encode a variety of file system semantics. The challenge of file system verification using the SMT solver was to translate reachability property because the SMT solver does not support any built-in transitive closure operator. We employed the tactics and strategies proposed in [28] and [16] to incorporate the reachability property in the context of the SMT solver. Table 2 summaries the results of verification of four file system semantics and the time taken by the tool. The tool was run on a Mac Mini, 3 GHz Intel Core i7. The number of operations is given without taking into account operation arguments. The number of tokens specifies the number of operations that require synchronization. The analyzer shows that the concurrent execution of move operations is anomalous, i.e., it may violate the tree invariant. It follows that no file system can support an unsynchronised move without anomalies, such as loss or duplication.

8 Related work

First-Order Logic Reasoning. A number of formalisations of file systems have been proposed using first-order logic [6, 17, 23]. Most of them focus only on primitive file I/O operations, such as reading and writing file content [6, 29]. Arkoudas et al. [6] have proved the correctness of read and write operations for a basic file system implementation using Athena, an interactive theorem prover. Given a simple file system implementation, Athena constructs 283 lemmas and

theorems in order to verify the isolation of reading and writing files in a directory. Hughes [24] has specified a visual file system using the Z notations [44]. He focuses on modelling of a hierarchical file system, so that its model covers basic operations affecting the tree structure, including move and remove directories. However, its specification does not consider the no-loop property, it only takes transitive closure (i.e., reachability) as the main property of a tree structure. Inspired by Hughes's specification, Damchoom et al. [14] have formalised and proved a tree-structured file system by using Event-B and Rodin platform [5]. Like our specification, their model is based on acyclic directory structure. A set of permissions are attached to an object, so that accesses to the object depend on the permissions allowed. The Rodin toolset generates 162 proof obligations to verify the specification model. Hesselink and Lali [23] have introduced an alternative approach to formulate the file structure using partial functions from paths to data.

However, the first-order logic reasoning does not scale well when reasoning about operation executions of a Posix file system [32]. The Posix English specification defines a set of preconditions for each operation, which must be satisfied before its execution. For instance, moving a source directory into a destination directory takes effect, if the source directory is not an ancestor of the destination directory. Encoding such conditions using first-order logic entails many proof obligations and constraints that increase non-linearly with respect to the size of programs [32].

Separation Logic Reasoning. Recent work on file system verification relies on separation logic [37]. Haogang et al. [22] have introduced Crash Hoare Logic (CHL) for developing and verifying sequential and fault-tolerant file systems. The CHL logic checks whether a storage system implementation will recover to a state consistent with its specification after a failure. Using the analysis, the authors specified and verified FSCQ, a crash-safe user-space file system implemented in Haskell. The FSCQ's interface consists of a series of Hoare triples over high-level operations. The specification model of FSCQ relies on the separation logic to reason about operations at different level of abstractions including disk, files, directories, and logical disk. FSCQ uses a write-ahead log for failure recovery. The CHL analysis proved that the write-ahead log guarantees atomicity of updates by adding fault-conditions into the Hoare triples.

Biri and Galmiche [10] have proposed a separation logic rule for trees and local reasoning over global paths. However, their simple tree model forbids structural modifications, as neither new nodes can be created nor nodes can be moved, i.e., the tree structure is static.

Gardner et al. [18] have proposed a formal model of Posix file system based on separation logic. The semantics of Posix operations are captured with preconditions and postconditions in a Hoare-logic style. Some permissions are associated into each operation to control access to shared paths. Before applying an update, the necessary permissions must be obtained in order to ensure that the effect of the update is propagated to entries whose path may overlap. However, the specification model does not support concurrent Posix users.

Conflict Resolution in File systems. Clements et al. [13] have proposed a cache conflict-free implementation of Posix file system on a shared-memory multiprocessor system. They explore the commutativity of Posix operations to design a scalable file system implementation. They have presented an analyser, called COMMUTER, which checks the commutativity of Posix operations. COMMUTER relies on symbolic executions for program testing. A symbolic model tests all permutations of operations, and computes necessary conditions under which those operations commute. Using the commutativity conditions, they modify the Posix semantics. COMMUTER generates different test cases to verify the semantics in a real implementation. However, they focus only on scalability, not on the safety of executing commutative operations; they do not verify that the commutative operations maintain the tree invariant.

Balasubramaniam and Pierce [8] have proposed an optimistic files system replication model from a semantics perspective. Causally-dependent operations are ordered according to a happen-before relation, while concurrent operations may be executed in any orders. Concurrent updates on the same directory are allowed if they do not conflict. For instance, concurrent users can add different files with different names to the same directory, but if one user modifies a file, and another deletes its parent directory, a conflict happens. The model requires users to manually resolve conflicts. This specification model was later formalised and proved by Ramsey and Csirmaz [36]. However, the operation-based model is limited i.e., the algebra model contains 51 different rules for few operations, including create, remove, and edit. It is not clear how one can extend the model to support more complex operations, such as move operations involving different directories. In addition, the model does not check the tree invariant; it is difficult to describe the acyclic property by using their model.

Bjϕrner [11] has proposed a replicated file system reconciler (DFS-R) that automatically resolves conflicts when they arise. The author uses model checkers to verify the conflict resolution strategies. Similarly to the CISE-enabled tool, the analysis gives a counter-example for concurrent moves, meaning that concurrent move operations do not maintain directory hierarchies as tree-like structure. However, this reconciler does not address how to add synchronisation when the tree invariant is violated. In this vein, Microsoft One Drive [4] discards the directories involved, thus restoring a tree. Similarly, Google Drive [3] moves the involved directories directly under the root. The geoFS [45] system effectively executes a problematic move as copy-delete, duplicating the directories involved.

9 Conclusion and Future Work

We have applied the CISE analysis to verify and co-design an available file system. Initially, the file system specification models the POSIX file system. The main invariant is that the file system structure must be shaped as a tree. We verified that our co-design approach is able to remove synchronisation for the common file system operations, while ensuring the tree invariant.

There are several avenues for future work from both verification and performance perspective. First, the CISE analysis only verifies the correctness of the file system against concurrent executions. In the future,we plan to propose proof rules that allow developers to reason about the operation executions in the presence of replica and network failures. Thus, programmers would be able to prove that a file system specifications model handles properly any possible faults. This entails formalisation of failure models, as the specification of the file-system API captures its semantics under crashes. We are going to implement the three file system semantics to compare their actual performance under real workloads. The plan is to integrate our co-design findings into a highly-scalable geo-replicated file system. The challenge is to translate the tokens into an efficient concurrency control protocol, which is also dead-lock free. We are looking for dynamic and heuristic analysis that allow to measure and improve the token implementations.

Bibliography

[1] Https://github.com/Z3Prover/z3
[2] POSIX.1-2008. The Open Group Base Specifications Issue 7
[3] Google Drive (2017), https://www.google.com/drive/
[4] Microsoft OneDrive (2017), https://onedrive.live.com/
[5] Abrial, J.R.: A system development process with event-b and the rodin platform. In: Proceedings of the Formal Engineering Methods 9th International Conference on Formal Methods and Software Engineering (ICFEM). pp. 1–3. Berlin, Heidelberg (2007)
[6] Arkoudas, K., Zee, K., Kuncak, V., Rinar, M.: Verifying a file system implementation. In: Proceedings of the Formal Methods and Software Engineering. pp. 373–390. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
[7] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K.: Measurements of a distributed file system. In: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles. pp. 198–212. SOSP '91, ACM, New York, NY, USA (1991)
[8] Balasubramaniam, S., Pierce, B.C.: What is a file synchronizer? In: Int. Conf. on Mobile Comp. and Netw. (MobiCom '98). ACM/IEEE (Oct 1998)
[9] Bernstein, P., Radzilacos, V., Hadzilacos, V.: Concurrency Control and Recovery in Database Systems. Addison Wesley Publishing Company (1987)
[10] Biri, N., Galmiche, D.: Models and separation logics for resource trees. Journal of Logic and Computation 17(4), 687–726 (2007)
[11] Bjørner, N.: Models and software model checking of a distributed file replication system. In: Formal Methods and Hybrid Real-Time Systems. pp. 1–23 (2007)
[12] Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference. pp. 28–28. ALS'00, Berkeley, CA, USA (2000)
[13] Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., Kohler, E.: The scalable commutativity rule: Designing scalable software for multicore processors. In: Symp. on Op. Sys. Principles (SOSP). pp. 1–17. ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery, Farmington, PA, USA (2013)
[14] Damchoom, K., Butler, M., Abrial, J.R.: Modelling and proof of a tree-structured file system in event-b and rodin. In: Proceedings of the 10th International Conference on Formal Methods and Software Engineering. pp. 25–44. ICFEM '08, Springer-Verlag, Berlin, Heidelberg (2008)
[15] Davidson, S.B., Garcia-Molina, H., Skeen, D.: Consistency in a partitioned network: a survey. ACM Comput. Surv. 17(3), 341–370 (Sep 1985), http://doi.acm.org/10.1145/5505.5508
[16] El Ghazi, A.A., Taghdiri, M.: Analyzing alloy constraints using an smt solver: A case study. In: 5th International Workshop on Automated Formal Methods (AFM). Edinburgh, United Kingdom (2010)

[17] Freitas, L., Woodcock, J., Butterfield, A.: Posix and the verification grand challenge: A roadmap. In: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 153–162 (March 2008)

[18] Gardner, P., Ntzik, G., Wright, A.: Local reasoning for the posix file system. In: Shao, Z. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 8410, pp. 169–188. Springer Berlin Heidelberg (2014)

[19] Ghemawat, S., Gobioff, H., Leung, S.T.: The Google File System. In: Symp. on Op. Sys. Principles (SOSP). pp. 29–43. Assoc. for Computing Machinery, Bolton Landing, NY, USA (Oct 2003)

[20] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In: Symp. on Principles of Prog. Lang. (POPL). St. Petersburg, FL, USA (2016)

[21] Guy, R., Heidemann, J.S., Mak, W., Popek, G.J., Rothmeier, D.: Implementation of the ficus replicated file system. In: USENIX Conference Proceedings. pp. 63–71 (1990)

[22] Haogang, C., Daniel, Z., Tej, C., Adam, C., Frans, K.M., Nickolai, Z.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. SOSP '15, ACM, New York, NY, USA (2015)

[23] Hesselink, W.H., Lali, M.: Formalizing a hierarchical file system. Formal Aspects of Computing 24(1), 27–44 (2010)

[24] Hughes, J.: Specifying a visual file system in z. In: IEEE Colloquium on Formal Methods in HCI: II. pp. 3/1–3/3 (Dec 1989)

[25] Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. North-Holland (1983)

[26] Kistler, J.J., Satyanarayanan, M.: Disconnected operation in the Coda file system. In: Symp. on Principles of Dist. Comp. (PODC). vol. 10, pp. 3–25 (Feb 1992)

[27] Kumar, P., Satyanarayanan, M.: Flexible and safe resolution of file conflicts. In: Usenix Tech. Conf. New Orleans, LA, USA (Jan 1995)

[28] Leino, K.R.M.: Automating induction with an smt solver. In: Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 315–331. VMCAI'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-27940-9_21

[29] Morgan, C., Sufrin, B.: Specification of the unix filing system. vol. SE-10, pp. 128–142 (1984)

[30] Nadkarni, A.: Scale-out file systems on object-based storage platforms. IDC Technology Assessment 258393, International Data Corporation (IDC), Framingham, MA, USA (2015)

[31] Najafzadeh, M., Gotsman, A., Yang, H., Ferreira, C., Shapiro, M.: The CISE tool: Proving weakly-consistent applications correct. In: W. on Principles and Practice of Consistency for Distributed Data (PaPoC). EuroSys 2016 workshops, ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery, London, UK (Apr 2016)

[32] Ntzik, G., Gardner, P.: Reasoning about the posix file system: Local update and global pathnames. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 201–220. OOPSLA 2015, ACM, New York, NY, USA (2015)

[33] Ousterhout, J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M., Thompson, J.G.: A trace-driven analysis of the unix 4.2 bsd file system. In: SIGOPS Oper. Syst. Rev. vol. 19, pp. 15–24. ACM, New York, NY, USA (Dec 1985)

[34] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., Hitz, D.: NFS version 3 design and implementation. In: Proceedings of the Summer 1994 USENIX Technical Conference. pp. 137–152 (1994)

[35] Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Symp. on Op. Sys. Principles (SOSP). pp. 288–301. ACM SIGOPS, Saint Malo (Oct 1997)

[36] Ramsey, N., Csirmaz, E.: An algebraic approach to file synchronization. Tech. Rep. TR-05-01, Harvard University Dept. of Computer Science, Cambridge MA, USA (May 2001)

[37] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. LICS '02, IEEE Computer Society, Washington, DC, USA (2002)

[38] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.: Design and implementation of the Sun Network Filesystem. In: Summer 1985 USENIX Conf. pp. 119–130. USENIX, Portland OR, USA (Jun 1985)

[39] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. In: IEEE Trans. on Computers. vol. 39, pp. 447–459 (1990)

[40] Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies. FAST '02, USENIX Association, Berkeley, CA, USA (2002)

[41] Schwan, P.: Lustre: Building a file system for 1,000-node clusters. In: Proceedings of the 2003 Linux Symposium (2003)

[42] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS). Lecture Notes in Comp. Sc., vol. 6976, pp. 386–400. Springer-Verlag, Grenoble, France (Oct 2011)

[43] Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–10. MSST '10, IEEE Computer Society, Washington, DC, USA (2010)

[44] Spivey, J.M.: The z notation: A reference manual. In: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (1998)
[45] Tao, V., Shapiro, M., Rancurel, V.: Merging semantics for conflict updates in geo-distributed file systems. In: ACM Int. Systems and Storage Conf. (Systor). pp. 10.1–10.12. Haifa, Israel (May 2015)
[46] Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani: A scalable distributed file system. In: SIGOPS Oper. Syst. Rev. vol. 31, pp. 224–237. ACM, New York, NY, USA (Oct 1997)
[47] Vogels, W.: File system usage in windows nt 4.0. In: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP). pp. 93–109. ACM, New York, NY, USA (1999)
[48] Vogels, W.: Eventually consistent. In: ACM Queue. vol. 6, pp. 14–19 (Oct 2008)
[49] Wang, A.I., Reiher, P., Bagrodia, R., Kuenning, G.: Understanding the behavior of the conflict-rate metric in optimistic peer replication. In: Proceedings of the 13th International Workshop on Database and Expert Systems Applications. pp. 757–761 (Sept 2002)

Abstraction-Based Interaction Model for Synthesis

Hila Peleg[1], Shachar Itzhaky[1], and Sharon Shoham[2]

[1] Technion, {hilap,shachari}@cs.technion.ac.il
[2] Tel Aviv University, sharon.shoham@gmail.com

Abstract. Program synthesis is the problem of computing from a specification a program that implements it. New and popular variations on the synthesis problem accept specifications in formats that are easier for the human synthesis user to provide: input-output example pairs, type information, and partial logical specifications. These are all partial specification formats, encoding only a fraction of the expected behavior of the program, leaving many matching programs. This transition into partial specification also changes the mode of work for the user, who now provides additional specifications as long as they are unhappy with the result. This turns synthesis into an iterative, interactive process.
We present a formal model for interactive synthesis, leveraging an abstract domain of predicates on programs in order to describe the iterative refinement of the specifications and reduction of the candidate program space. We use this model to describe the behavior of several real-world synthesizers. Additionally, we present two conditions for termination of a synthesis session, one hinging only on the properties of the available partial specifications, and the other also on the behavior of the user. Finally, we show conditions for realizability of the user's intent, and show the limitations of backtracking when it is apparent a session will fail.

1 Introduction

Program synthesis is the problem of computing from a specification a program that implements it. The classic synthesis problem searches for an implementation to a full specification, usually encoded in some logic. Newer variations on the problem have turned to partial specifications, such as input-output examples or type information, that are easier for the user to provide but describe a much wider array of matching programs.

Synthesis tools for end-users are available for a wide variety of purposes from creating formulae in Microsoft Excel [10] to formulating SQL queries [34]. Programming by Example tools that accept input-output pairs as their specification have also matured enough to be practical on their own [2, 10, 17, 18, 20, 23, 34, 35, 36] or as a way to refine the results of type-driven synthesis [7, 24].

When the specifications are partial, the user is often brought into the loop to aid the synthesizer to determine the correctness of the final product and to direct it with additional feedback in case of ambiguity. Gulwani [11] separates synthesizers by their model of interaction with the user. Notably two categories are (i) *user-driven* synthesis tools, in which the user is responsible for verifying the artifact returned by the synthesizer, and if incorrect, for providing additional specifications to the synthesizer, and (ii) *synthesizer-driven* tools, in which the synthesizer poses the user with membership queries for ambiguous examples until it has reached a level of confidence high enough

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 382–405, 2018.
https://doi.org/10.1007/978-3-319-73721-8_18

to return a program to the user as a validation query. Counterexample-guided Inductive Synthesis (CEGIS) [33], in which a verifier is provided with a specification, and each program from the synthesizer is verified to produce either acceptance or an automatically generated counterexample, is seen as its own category, as the interactivity is between the synthesizer and the verifier.

Interactive synthesis Despite the fact that few user-driven tools define themselves as interactive synthesis tools, it is important to note that interactivity is always inherent in the synthesis workflow: the user provides some initial specification, runs the synthesis procedure, and is presented with an answer. However, they may not be satisfied with this answer, which leads to refinement of the specifications and another execution of the synthesizer. This iterative process of candidate solution and refinement is rarely discussed, as focus tends to remain on each single attempt to reach the user's intended program with as partial a specification as possible, via rankings and biases.

Interaction via predicates on programs Likewise, while each synthesis tool usually treats the mode of specification it leverages as its own domain—input-output examples, types, etc.—the common ground is often overlooked. Each of these modes of feedback can be seen as a predicate over programs, and the process of providing a partial specification as constraining the space of possible programs to just those that satisfy each of the predicates. For instance, an input-output pair (i, o), often seen as the simple and natural tool for partial specification, can be seen as the predicate $[\![program]\!](i) = o$. As previous work [4, 26] has shown, examples are a weak tool with which to provide specification. The addition of other predicates in works such as [1, 24, 26] allows for better separation between programs.

The most comprehensive specification that describes a target program m^* is every predicate available in the system that holds for m^*. However, this complete specification is very likely infinite or not computable. On the other hand, an initial specification can be so partial as to rule out only a small number of programs in the candidate space. The solution is to leverage the user's ability to compute what is incomputable. In the words of Knuth, "Some tasks are best done by machine, while others are best done by human insight; and a properly designed system will find the right balance" [15] — by incrementally providing additional predicates to refine the specification, a process which at the limit will reach the comprehensive specification. In this paper, we limit our scope to this form of iterative synthesis, where the set of predicates is built monotonically.

Properties of iterative synthesis The attempts to get program synthesis tools to return a "suitable" program as soon as possible are based in heuristics and optimizations, and vary greatly from one tool to the next. Additionally, these tools often have no concept of convergence, or when the session will be forced to end, to explain their behavior should these heuristics fail. While a few of the tools have been modeled individually, in order to prove specific properties, the field is still lacking a generic model that can be used to prove properties such as termination and establish criteria for the designers of future synthesis tools to take into account as they design their new frameworks.

Goal The goal of this work is to investigate the theoretical foundation of interactive synthesis. To this end, we present a model of the iterative synthesis process, centered around the interaction between the synthesizer and a human user, and grounded in the

theory of abstract domains [3]. This model aims to capture work with a wide array of user-driven synthesizers. We use this model to prove both existing properties of synthesizers and desirable properties in future synthesis tools. In order to do so, our definitions and results are grounded in real-world examples. This model provides us with a theoretical understanding of the properties of the interaction (e.g, progress, termination guarantees) which can then be applied to current and future synthesizers.

Existing work Previous work has modeled single iterations of different flavors of synthesis [1, 29], and the counterexample-guided model of synthesis (CEGIS) [14, 31]. The synthesizer-driven model of program synthesis [21] has also been modeled via predicates, where user answers to membership queries are translated into constraints and used to reduce the search space for the next iteration. A learner-teacher model of program synthesis [22] has been presented mainly to model CEGIS, but can be applied to an iterative, user-driven synthesizer as well, with the human user taking on the role of the teacher. However, this model provides only guarantees stemming from the properties of the program space made available by the synthesizer, with little consideration of the way feedback is provided to the synthesizer. For a CEGIS model, this is sufficient, as communication between the teacher and learner is chiefly in examples, but is unsuitable for a more generic model where feedback formats and specification tools are multiform.

1.1 Our approach

In this paper, we formulate a model for interactive synthesis using the theory of abstract domains.

An abstract domain of predicates Given a domain of programs M and a domain of predicates on programs $\mathcal{P}$, we define the concrete domain of the synthesis algorithm to be sets of programs $(2^M, \subseteq)$ and the abstract domain to be sets of predicates $(2^\mathcal{P}, \supseteq)$, with an abstraction function that produces the incomputable set of all predicates that hold for the set of programs, and a concretization function that produces the equally incomputable set of all programs that satisfy a conjunction of all predicates in a given set. Since both these sets are likely not computable, a real synthesizer relies on the synthesizer's representation of the state to replace a concretization, and the user to replace the abstraction. Section 3 formally defines these domains and the operations on them.

Iterative, interactive synthesis In this domain, we can then define an iterative synthesis algorithm as an iterative refinement (i.e., adding of predicates) of the specification in each iteration of the process. This creates a synthesizer state, in itself an abstract element, from which the next program displayed to the user as a candidate solution is selected. This process, in essence, is leveraging the user to compute the abstraction of the target program, or more accurately, a finite subset of it. If a finite subset that underapproximates the target exists, the synthesis session can converge regardless of the implementation. Section 4 defines an iterative synthesis session, the notion of progress by the user, and the terms for convergence.

Properties of interactive synthesis Using this model, we show several properties of interactive synthesis. In section 6 we define the point from which a synthesis session can no longer converge, even if the user has, from their point of view, only provided correct specifications, and properties of the point we must backtrack to when that happens.

Section 5 offers two separate sets of limitations on the model that lead to convergence (i.e., a finite session) in every session. A *well-quasi-order of predicates* ensures that all sessions will terminate, and a *locally strongest user* condition ensures termination when predicates only have a well-founded-ordering. We demonstrate these conditions and properties using realistic examples.

Implications Finally, section 7 discusses the implications of these properties for the designers of future synthesis tools.

1.2 Main contributions

The main contributions of this paper are:

– A general model for iterative synthesis using the theory of abstract domains,
– Convergence conditions for iterative synthesis sessions, based on properties of the predicates and user behavior,
– Insights about backtracking when a session can no longer converge, and
– Recommendations for designers of future synthesis tools.

2 Background

In this work we address program synthesis. Below we provide some background on synthesis methods and terminology.

The synthesis problem description Readers familiar with software verification would most likely recognize the common verification problem $\forall \iota.\ \varphi(\iota)$, where ι ranges over possible program inputs and φ is a property to check (safety, liveness, termination, etc.). In synthesis, the problem is commonly stated as $\exists m.\ \forall \iota.\ \varphi(m, \iota)$, where m ranges over the domain of *candidate programs*, and the synthesizer is tasked with finding one program that satisfies the desired property on all inputs. Different tools have varying ways to define the candidate program space. Since this space is huge even when considering a modest program size, sifting through it to find a single program with the property φ is computationally hard.

Program semantics In the most abstract sense, a program m accepts input ι and produces output ω. In programs that have effects on their environment (sending network packets, moving a robotic arm) the environment state can be folded into the input and output spaces; so for all purposes, we can assume a definition of program semantics as $[\![m]\!] : I \to O \cup \{\bot\}$. The special value $\bot$ indicates abnormal behavior, which may be a run-time error or non-termination. It means there is no execution of the program with the given input that reaches the designated "successful" exit point.

Partial specification Often, it is quite hard to describe the property φ to the synthesizer precisely. Most synthesizers offer a domain-specific language for describing weaker properties in a way that both the user and the synthesizer can understand and (hopefully) the synthesizer can efficiently generate a corresponding program. We present two examples for such property domains.

– **Type-directed synthesis** is a sub-class of the synthesis problem where the specifications are in the forms of types: the types of the input variables and the expected output type. Likewise, the construction of programs is guided by derivation rules

that are constructed from the typing rules. On a very basic level, the construction of well-typed programs is type-directed synthesis, but synthesizers often contain derivation rules that select a small number of operations to derive when adding operations or assigning parameters.

This class can be restricted to our setting of monotonic refinement if at each step the user can only (a) remove variables from the scope, (b) generalize an input variable's type to a super-type, or (c) concretize the expected return type to a sub-type.

- **Programming by Example (PBE)** is a sub-class of the synthesis problem where all communication with the synthesizer is done using input-output examples. The classic PBE problem is defined as a set $\mathcal{E}$ of examples, each of which is a pair of an input and its corresponding expected output; the result is a program m, within the candidate space, that satisfies every example in $\mathcal{E}$. PBE has become widely popular since examples are easier to create than full logical specifications, can be provided in many formats from tabular data to unit tests, and can even be created by non-programmers. Since there might be more than one program m in the language $\mathcal{L}$ that matches all specifications, the iterative PBE problem was introduced. In the iterative model, each candidate program m_i is presented to the user, which may then accept m_i and terminate the run, or answer the synthesizer with additional examples $\mathcal{E}_i$ that direct it in continuing the search.

  As an extension of this approach, *abstract examples* can be used to describe a (possibly infinite) set of examples using a short description. This description usually uses a weak abstraction mechanism, such as regular expressions.

3 Foundations for Synthesis with Abstract Domains

In this paper, we formalize interactive synthesis using abstract domains, where the role of the user is to strengthen the abstraction of the target program, while the role of the synthesizer is to concretize the abstraction and pick a concrete element from it as a candidate program. To do so, we start, in this section, by formalizing a single iteration that consists of a user providing a spec as input and the synthesizer returning a program.

Let us consider U the domain of all programs, in all languages. Out of these, only a subset is available to the user via the synthesizer. We denote this, our program search space, $M \subseteq U$.

User-driven synthesis is guided by the concept of a target program in the user's mind. We denote $U^* \subseteq U$ the set of programs that satisfy the user's concept of a correct program, and $M^* = U^* \cap M$, the subset of U^* that is in the synthesizer's search space. A user's intention is *realizable* if $M^* \neq \emptyset$. It is important to notice for the remainder of this paper that M^*, while a subset of the synthesizer's search space, is not actually known to the synthesizer.

In order to encode the specification, let us also consider a (possibly infinite) set $\mathcal{P}$ of predicates over programs. We assume that every $p \in \mathcal{P}$ is decidable. When considered against some set of programs T, each predicate $p \in \mathcal{P}$ defines a subset of programs from T that satisfy it, denoted $\{m \in T \mid m \models p\}$. In this way, the same set of predicates $\mathcal{P}$ can define subsets of both M and U. In this sense, the predicates can be viewed as formulas, and the programs as structures. We do not, however, assume or use any internal structure of the predicates in this paper.

In particular, we will use predicates in implication modulo a theory of programs. We write $p \Rightarrow_T q$ to denote $\forall m \in T.\ m \vDash p \Rightarrow m \vDash q$. The same extends to a set of predicates, $A \Rightarrow_T q$, to mean as their conjunction.

The remainder of this paper assumes working with a specific $\mathcal{P}$ and a specific M, and that the user is seeking a specific M^*. This means all the definitions that follow are parametric in M and $\mathcal{P}$, and when used also in M^*.

3.1 An abstract domain for programs

Our concrete domain consists of the powerset lattice $(2^M, \subseteq)$ (where the least element is $\emptyset$ and the greatest element is M). That is, each concrete element is a set of programs, and the sets get smaller when lower in the lattice.

During the synthesis process, the synthesizer represents (or abstracts) sets of programs from the concrete domain using sets of predicates from $\mathcal{P}$. Formally, let $\mathcal{A} = 2^{\mathcal{P}}$. The synthesis process uses an abstract domain that consists of the powerset lattice $(\mathcal{A}, \sqsubseteq)$, where $\sqsubseteq$ is defined as $\supseteq$. That is, each abstract element is a set of predicates (interpreted as a conjunction), and the sets get larger (or more constrained) when lower in the lattice. Join, meet, bottom, and top are defined as they usually are in the powerset domain: For two abstract elements $A_1, A_2 \in \mathcal{A}$, meet is defined as $A_1 \sqcap A_2 = A_1 \cup A_2$ and join as $A_1 \sqcup A_2 = A_1 \cap A_2$. Further, $\top = \emptyset$ and $\bot = \mathcal{P}$.

From here on, we refer to $A \in \mathcal{P}$ as elements in the lattice and as sets of predicates interchangeably. Which one we mean should be clear from the context (e.g.., the operators used).

Galois connection We would like an abstract element $A \in \mathcal{A}$ to represent the set of programs $s \in M$ for which every predicate $p \in A$ holds. To do so, we define a Galois connection between $(2^M, \subseteq)$ and $(\mathcal{A}, \sqsubseteq)$.

Definition 1 (Abstraction). *For a single program $m \in M$, we define the* abstraction *function $\beta(m) = \{p \in \mathcal{P} \mid m \vDash p\}$, which abstracts m into the set of all predicates that hold for m. From this we define for a set of programs $C \subseteq M$ the abstraction $\alpha(C) = \bigsqcup_{m \in C} \beta(m) = \{p \in \mathcal{P} \mid \forall m \in C.\, m \vDash p\}$.*

This is similar to the abstraction performed by Houdini [8], Daikon [5], and D^3 [25].

Definition 2 (Concretization). *For an abstract element $A \in \mathcal{A}$, we define the* concretization *function $\gamma(A) = \{m \in M \mid \forall p \in A.\ m \vDash p\}$, or all programs for which all constraints in A hold.*

It is easy to verify that (α, γ) is a Galois connection.

Recall that in the abstract domain, $\bot = \mathcal{P}$ and $\top = \emptyset$. Therefore, $\gamma(\top) = M$, which means that the top element represents all valid programs in M, as desired. On the other hand, $\gamma(\bot)$ is not necessarily the empty set, since there might be valid programs that satisfy all predicates in $\mathcal{P}$. However, in the typical case, $\mathcal{P}$ contains contradicting predicates (e.g., a predicate and its negation, or examples mapping the same input i to different outputs $o_1 \neq o_2$), in which case $\gamma(\bot)$ represents an empty set of programs.

Reducing the search space The non-interactive, single-step, synthesis problem can now be described as one for which the input is a (partial) specification of the target program in the form of an abstract element $A \in \mathcal{A}$, and the output is some program

from the set of programs it describes. The selection is (usually) not random, but rather influenced by internal representation in the synthesizer, as well as ranking functions. To reason about the synthesizer's role, we define $Select : \mathcal{A} \rightarrow M \cup \{\bot\}$, the synthesizer's operation of finding such a program. $Select(A)$ amounts to picking a concrete element from $\gamma(A)$, or returning $\bot$ if no such element exists; hence, it can be understood as partially concretizing the abstract element. The implementation of $Select$ is dependent on the synthesis algorithm being used.

3.2 Examples

Type-directed synthesis as an abstract domain A widely used domain of predicates is a domain of type information. When creating a procedure via type-directed synthesis, the specification to the synthesis procedure is provided via type predicates for the procedure's formals $(name, \tau) \in Formals \times \mathcal{T}$ and a desired return-type predicate, $\tau_{ret} \in \mathcal{T}$ which will hold according to the $\sqsubseteq$ relation on types. A similar specification is used for type-directed synthesis that produces code snippets: the same τ_{ret} specifies the target type (usually assigned to a variable) and the available variables are specified using type predicates $(name, \tau) \in Vars \times \mathcal{T}$ for local variables $Vars$.

PBE as an abstract domain Another frequently used domain of predicates is the domain of input-output examples. Recall that each program m defines a function, $[\![m]\!] : I \rightarrow O \cup \{\bot\}$, that maps inputs to outputs (or to error). Programming by example considers the predicates $\mathcal{P} = I \times O$, where each pair $(\iota, \omega) \in \mathcal{P}$ dictates that for input ι, the program outputs ω. For this purpose, we define $m \vDash (\iota, \omega) \iff [\![m]\!](\iota) = \omega$.

Syntactic feedback as an abstract domain [26] introduces a domain of predicates that provide syntactic restrictions on programs, intended for use by programmers. For instance, an `include(f)` predicate which holds only for programs that make use of a function or operator f, or `exclude(f)` which holds only when they do not. For linear functional programs, these operators can also be generalized to sequences of methods, either as a continuous subsequence—`exclude(`$f \cdot g$`)` will hold only for programs where f is not immediately followed by g—or for general subsequences—`exclude(`$f \cdot g$`)` will hold for programs where there are no $i < j$ s.t. $f_i = f, f_j = g$. The predicates used in this work are limited to these, but in several examples in this paper we make use of predicates suggested by or simply in the spirit of those shown in [26].

3.3 Computability of the model

We notice that, in general, both α and γ are non computable: α because $\mathcal{P}$ may be infinite; and even though any A provided by the user will always be a finite set, γ may still not be computable as a finite set of predicates may return an infinite subset of an infinite M. Because of that, neither of them is used directly by any concrete implementation of the model. Concretization is only performed as part of a $Select(A)$ operation, representing the synthesizer's generation of a program based on its description of the reduced program space A, which need not actually create the concrete set of programs represented by A. In synthesizers based on version space algebra (VSA) [19], for instance, only a representation of the space of all programs is constructed, from which a single concrete program is then selected.

Abstraction is also never performed by the algorithm, but rather by the user: the target programs, M^*, as envisioned by the user, are described in the input specification A by the selected predicates. This is less precise than a full (and possibly infinite) $\alpha(\{m^*\})$ of some $m^* \in M^*$, but in an iterative synthesis process can be refined by the user when the result is insufficient, which means that the synthesizer state (representing the accumulated user input) comes closer to $\alpha(\{m^*\})$ with each iteration. (Note that unlike a classical abstraction framework [3], where it is important to soundly abstract the entire set M^*, in synthesis it suffices to abstract some nonempty subset of M^*.)

Intuition If the user could produce a full specification $S^* \subseteq \mathcal{P}$ (or as full as $\mathcal{P}$ allows), satisfying it could be a matter of arbitrarily selecting any program from $\gamma(S^*)$. However, since creating full specifications is hard or even impossible, the process of interactive synthesis, which will be described in the next section, is essentially building up to a fuller specification in every iteration. The user adds new specifications to rule out each undesirable candidate program, and the meet operation collects added specifications into the synthesizer state, which at the limit will reach S^*.

4 An Abstract Model of Interactive Synthesis

Section 3 discussed a model for a single iteration of synthesis. We now wish to describe the iterative process that exists, even if implicitly, in most synthesizers. In it the user will keep adding to the specifications given every time the synthesis procedure offers an unsatisfactory candidate. We formulate this as questions (candidate programs) and answers (additional specifications).

Definition 3 (Synthesis session). *A* synthesis session *is a sequence of steps by the user and synthesizer $\mathcal{S} = (A_0, q1), (A_1, q_2), \ldots$ such that $q_i \in M \cup \{\bot\}$ are synthesizer questions and $A_i \in \mathbb{P}_{fin}(\mathcal{P}) \cup \{\bot\}$ are user answers, where $\mathbb{P}_{fin}(\mathcal{P})$ is the set of all finite subsets of $\mathcal{P}$ and $\bot$ signifies a forced contradiction. We denote A_0 the* initial specifications *provided by the user.*

Within a synthesis session we define the state of the synthesizer via the constraints on it provided by the user, as follows:

Definition 4 (Synthesizer state). *The* state of the synthesizer $S \subseteq \mathcal{P}$ *is an abstract element describing the portion of the search space requested by the user. If the user has given feedback for i iterations in the form of the elements $A_0, A_1, \ldots, A_i \subseteq \mathcal{P}$, the state after i iterations of feedback is $S_i = \bigcup_{0 \leq j \leq i} A_j$.*

Interactive synthesis can now be formalized as a process in which both the state of the synthesizer and the interaction between the synthesizer and the user are based on abstract elements. In step i, the synthesizer selects a program $q_i \in M$ using $Select(S_{i-1})$, and poses q_i as a validation question to the user. The user accepts or rejects the program. In case of rejection, the user responds with an answer $A_i \in \mathcal{A}$ in the form of an abstract element which consists of one or more predicates out of $\mathcal{P}$. Given the user's answer A_i, the new state of the synthesizer in step $i+1$ is set to $S_{i+1} = S_i \sqcap A_{i+1}$, thus narrowing the set of concretizations to consider. Or, in other words, we can now define $S_{i+1} = \bigsqcap_{0 \leq j \leq i+1} A_j$. The search is over either when $Select$ returns nothing because $\gamma(S_i) = \emptyset$ and represents no programs, or when the user is satisfied and accepts the program.

Notice that, unlike the classical use of abstraction, where the intent is to describe as many concrete states as possible, and so new information is appended via join, here our purpose is to refine, and so we use meet.

Synthesis users In order to reason about iterative synthesis, we must define the user's behavior. We have already defined U^* the set of programs in U the user is willing to accept, as well as M^*, the intersection between the user's concept of the target program and the search space of the synthesizer. We now add guarantees for the iterative behavior:

Definition 5 (User correctness). *A user step, providing A_i as an additional specification, is* correct *when $A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*.m \vDash p\}$.*

Correctness means the user will not provide predicates that are inconsistent with their idea of the target. Notice that this set of predicates may still contain a contradiction, as it contains predicates of different programs, and that even if no explicit contradiction exists, subsets of it may still evaluate to $\emptyset$ over the domain M.

According to definition 5, a correct user may still provide predicates that hold for some, but not all, of U^*. This may seem unintuitive, but realistically occurs because (a) U^* may not be sufficiently described with predicates from $\mathcal{P}$, but a subset of it may, (b) given a current candidate program m, the user sets a trajectory for the synthesis procedure and makes local decisions that may rule out some programs in U^*, or conflict with other (similarly local) decisions made in the past.

Definition 6 (Synthesis user). *The* behavior of the user *includes the following guarantees:*

1. *The user is correct for as long as they can be. If the user can no longer provide an answer that is correct, they will answer $\bot$.*
2. *If a user sees a program in M^*, they will accept it.*

Finally we define a feasible synthesis session as a session that can be reached by the actions of a user and a synthesizer:

Definition 7 (Feasible synthesis session). *A* feasible synthesis session *is a synthesis session $\mathcal{S} = (A_0, q1), (A_1, q_2), \ldots$ that satisfies the following:*

(a) All A_i are correct steps (definition 5) or $\bot$,
(b) $q_i = Select(S_{i-1})$, i.e. $q_i \in \gamma(S_{i-1}) \cup \{\bot\}$, where $\bot$ signifies no possible program,
(c) If $q_n \in M^* \cup \{\bot\}$ then $\mathcal{S}$ is finite and of length n, and
(d) In a finite $\mathcal{S}$ of length n, $q_n \in M^* \cup \{\bot\}$

where item b is a requirements for synthesizer correctness, and items a, c and d are requirements for user correctness.

Remember that additionally, from the definition of $Select$, if $\mathcal{S}$ is finite of length n, then $q_n = \bot \iff \gamma(S_{n-1}) = \emptyset$.

These mean that a feasible synthesis session is either (i) infinite, (ii) ends by returning $\bot$, (iii) or ends with the user accepting q_n the last program.

For the remainder of this paper we are only interested in feasible synthesis sessions.

Definition 8 (Convergence). *A synthesis session* $(A_0, q1), (A_1, q_2), \ldots, (A_n, q_n)$ *is said to* converge *if* $\gamma(S_n) \subseteq M^*$. *It has* converged successfully *if* $\gamma(S_n) \neq \emptyset$.

When a session has terminated with any result other than $\bot$, this will mean that the user accepts q_n, but convergence is in fact a stronger condition. This is because definition 7(d) can refer to a case where the synthesizer has offered a program out of M^* at any point in the session, because of the implementation of $Select$, ranking, or domain knowledge, thereby causing the session to end immediately. Convergence, on the other hand, ensures that regardless of the implementation of $Select$, a program from M^* will be returned (or no program at all). This definition reflects the fact that, unlike classical abstraction frameworks, where one seeks an overapproximation of the target that is "precise" enough, convergence of a synthesis procedure requires an *underapproximation* of the target. For convergence to be successful, that underapproximation must be nonempty. For the remainder of this paper we will be mostly interested in the worst-case implementation of $Select$, where the session either converges or is infinite.

4.1 Progress-making sessions

The first basic property needed in order to explore convergence is that the synthesis session is progressing—refining not only the abstract element of the synthesizer state but also its concretization in the program space. We consider two kinds of progress, weak and strong, which differ by the effect of the step on the synthesizer state. Section 5 will leverage progress into results on termination.

Definition 9 (Weak progress). *A user answer* A_i *is said to create* weak progress *in iteration* i *of a synthesis session if* $\gamma(S_{i-1} \sqcap A_i) \subsetneq \gamma(S_{i-1})$. *This means that* A_i *has ruled out at least one program from* M *described by* S_{i-1} *.*

We say a synthesis session makes weak progress if every user answer A_i *in the session makes weak progress.*

Note that it is not enough to demand that $S_{i-1} \sqcap A_i \sqsupset S_{i-1}$: the user can provide a predicate p that rules out no program in $\gamma(S_{i-1})$, which means $\gamma(S_{i-1}) = \gamma(S_i)$ but since it was not given before by the user, $S_{i-1} \sqcap A_i \sqsupset S_{i-1}$.

Lemma 1 (Weak progress by implication). *User answer* A_i *in iteration* i *of synthesis session makes weak progress if and only if* $S_{i-1} \not\Rightarrow_M A_i$.

Proof. Let S be a synthesis session. Step i makes weak progress $\iff \gamma(S_{i-1} \sqcap A_i) \subsetneq \gamma(S_{i-1}) \iff \exists m \in M. m \in \gamma(S_{i-1}), m \notin \gamma(S_{i-1} \sqcap A_i) = \gamma(S_{i-1}) \cap \gamma(A_i) \iff \exists m \in M. m \in \gamma(S_{i-1}), m \notin \gamma(A_i) \iff \exists m \in M. \forall p \in S_{i-1}. m \vDash p, \exists p \in A_i. m \nvDash p \iff \exists p \in A_i. S_{i-1} \not\Rightarrow_M p \iff S_{i-1} \not\Rightarrow_M A_i$. □

Lemma 1 gives us a test for the synthesizer to apply should the creators of the synthesizer wish for it to enforce progress in every iteration.

Example 1. Let us examine predicates used for providing positive feedback. In PBE this might be an example that reinforces some behavior that is good in the current program. In other predicates this might be okaying a syntactic portion on the program, or in other words, asking the synthesizer to keep something for future programs. Another option

is approving of an intermediate value of the program for a specific input—something which holds for the current program.

All of these, while not ruling out the current program, may rule out other programs in the space. This means that in a synthesizer which enumerates the entire space of M in some order, the same q_i will be displayed as q_{i+1}. However, since the portion of the program space represented by S_n is different, some implementations of $Select$ may return a different program.

Definition 10 (Strong progress). *A user answer A_i is said to create* strong progress *in the synthesis session if $q_i \notin \gamma(S_{i-1} \sqcap A_i)$, or in other words, if $\alpha(\{q_i\}) \not\sqsubseteq A_i$.*

We say a synthesis session makes strong progress if every user answer in the session makes strong progress.

Definition 10 is stronger than that defined in definition 9 as it ensures the user will not be shown the same program again, regardless of the implementation of $Select$. If $Select$ has some preference bias—such as an ordering over the programs—then non-strong progress will essentially lead to the same program being returned; however, we do not preclude the general case where changing the specification in any way or even just re-running the synthesizer may yield a different program.

Example 2. The FlashFill implementation in Microsoft Excel [10] allows only predicates that would cause strong progress. Specifically, as the program candidate in each iteration of FlashFill is executed on the entire dataset and the results are shown to the user. The user can then make changes to records where the result of the executed program is not the desired result. This mean that the set of predicates available to the user at iteration i is not any $\{(r, o) \mid r \text{ is a record in the table}\}$, but only $\{(r, o) \mid [\![q_i]\!](r) \neq o\}$. Since every $p \in A_i$ necessarily rules out q_i, this is an even stronger requirement than that of strong progress in definition 10.

Due to our assumption on the user correctness, the strong progress requirement can be equivalently formulated by requiring the user to use at least one predicate that differentiates q_i from M^*:

Definition 11 (Diff). *We define the* diff *between two programs $m_1, m_2 \in M$ in the program space over the set of available predicates to be $diff(m_1, m_2) = \{p \in \mathcal{P} \mid m_2 \models p \wedge m_1 \not\models p\} = \beta(m_2) \setminus \beta(m_1)$.*

Lemma 2 (Correct strong progress by differentiating predicate). *A* correct *user answer A_i in iteration $i + 1$ of a synthesis session makes strong progress if and only if $A_i \cap \bigcup_{m \in M^*} diff(q_i, m) \neq \emptyset$.*

While progress is a natural requirement to make, it may not always be obtainable with the available predicates. There may simply not be predicates with which to rule out the current program, for instance, but, most often, there is simply no correct step with which to continue the session. Next, we define the result of the clash between progress and correctness and demonstrate a scenario where it manifests:

Definition 12 (Non-progress point). *Iteration i is a* weak non-progress point *(resp.* strong non-progress point*) if any predicate p that would cause weak (resp., strong) progress is incorrect, i.e., $\forall m \in U^*. m \not\models p$.*

In the sequel, we simply refer to a "non-progress point" since the weak/strong qualifier is determined by the kind of interaction enforced by the synthesizer.

If iteration i is a non-progress point, then by correctness the user is forced to answer $\bot$. In practice, this means iteration $i+1$ will necessarily be $(\bot_{\mathcal{P}}, \bot_M)$.

Example 3. Consider a domain of programs and a set of predicates $\mathcal{P} = \{\texttt{exclude(f)} \mid f \in \mathcal{V}\} \cup \{\texttt{include(f)} \mid f \in \mathcal{V}\}$ over some vocabulary of methods $\mathcal{V}$. The user is looking for a program that will provide them with the second element of a list of strings. Let us assume that $U^* = M^* = \{\texttt{input.tail.head}\}$, and that the user is shown $q_i =$`input.head.tail`.

If the current synthesizer enforces strong progress, the user is now at an impasse: includes are a form of positive feedback, approving of something in the current program. While they may rule out some program in the synthesizer state, they will not rule out q_i. However, with the given set of predicates, either option that will make any progress, `exclude(head)` and `exclude(tail)`, will violate correctness, and will cause $S_i \cap M^* = \emptyset$.

5 Termination

In general, a synthesis session may never terminate. For instance, it is easy to show using this model that PBE may never terminate: let us assume the user is searching for a program where conversion from polar to cartesian coordinates is required. The user will provide some examples for desired input-output pairs, and a program that applies the sine function to implement the conversion will be part of the synthesizer state, but no matter how many examples are provided, there will still be programs that use some interpolated polynomial instead of sine, thereby keeping $\gamma(S_i)$ from ever reaching M^*.

We now show two conditions for termination for synthesizers, based on properties of their predicates. The first is a condition for both strong and weak progress sessions, demanding a strong requirement from the synthesizer, a *well-quasi-ordering* of the predicates. The second is a condition for synthesis sessions that make strong progress, and is modeled on a property similar to well-quasi-order's finite basis property. In it, we can weaken the requirement on the predicates, but in exchange add a requirement from the user.

5.1 WQO predicates

We first show that termination can be guaranteed using the theory of well-quasi-ordering:

Definition 13 (Well-quasi-order [16]). *Let $\leqq$ be quasi-order on X (i.e., $\leqq \subseteq X \times X$ is a reflexive and transitive relation). By convention, $x > y$ denotes $y \leqq x \wedge x \nleqq y$. The following definitions are equivalent:*

(1) $\leqq$ is a wqo over space X
(2) In every infinite sequence $x_1, x_2, \ldots$ there exist $i < j$ s.t. $x_i \leqq x_j$, and
(3) X satisfies both: (a) every sequence $x_1 > x_2 > \ldots$ is finite (the strictly descending chain condition, also known as well-foundedness), and (b) every sequence $x_1, x_2, \ldots$ with $x_i \nleqq x_j$ for $i \neq j$ is finite (the incomparable chain condition, also known as the antichain condition).

Theorem 1. *Let* $p \preceq p' \iff p \Rightarrow_M p'$. *If* $\preceq$ *is a well-quasi-ordering over the set* $\bigcup_{m' \in M^*} \beta(m')$*, then any synthesis session that makes (weak or strong) progress will always converge in a finite number of steps.*

Proof. Since every strong progress session also makes weak progress, it suffices to prove the theorem for weak progress sessions.

Let us assume, by way of contradiction, that $\mathcal{S}$ is an infinite synthesis session that makes weak progress. We construct the infinite sequence $p_0, p_1, \ldots$ such that p_i is *some* progress-making predicate from A_i. Since $\mathcal{S}$ makes weak progress, we know that $S_{i-1} \not\Rightarrow_M p_i$ (Lemma 1) and in particular, for every $p' \in S_{i-1}$, $p' \not\Rightarrow_M p_i$. From definition 4, $\forall p_j . j < i \Rightarrow (p_i \not\Rightarrow_M p_j)$, or in other words, $\forall p_j . j < i \Rightarrow (p_i \not\preceq p_j)$. But since $\preceq$ is a wqo, in every infinite sequence $\exists i, j. i < j \wedge p_i \preceq p_j$ (from definition 13(b)), leading to a contradiction. This means a session must be finite, i.e. converge. □

From this, if the entire predicates set $\mathcal{P}$ is a wqo, then the synthesizer will terminate for every M^*.

Example 4. While it is easy to see that examples are not a wqo, as the entire domain is incomparable, there are domains of predicates that do create a wqo. For instance, a family of syntactic predicates $exclude(f_1 \cdots f_2 \cdots f_n)$ that exclude programs containing a specific subsequence of function calls (not necessarily consecutive) will be a wqo over the domain of linear programs [13]. In this domain, a user can express feedback such as $exclude(\texttt{close} \cdots \texttt{read})$, thereby ruling out every program that creates a read-after-close error.

5.2 Locally strongest user

In this subsection, we relax the well-quasi-order requirement on the predicates, and prove another termination property by assuming some locally-optimal property of the user.

Definition 14 (Base set). *Let* $S \subseteq \mathcal{P}$ *be a set of predicates. We define the* base *of* S, $Base(S) = \{p \in S \mid \forall p'. p' \Rightarrow_M p \Rightarrow p = p'\}$*, i.e. the set of strongest predicates in* S.

In order to simplify we assume $\mathcal{P}$ does not contain equivalent predicates.

Let us now add a new restriction on the user, which strengthens the strong progress requirement of the synthesizer:

Definition 15 (Locally strongest user). *Given a candidate program* $q_i \notin M^*$, *a* locally strongest user *will answer with* A_i *such that* $A_i \cap \bigcup_{m' \in M^*} Base(diff(q_i, m')) \neq \emptyset$. *That is, at least one predicate in the answer* A_i *will be taken from* $Base(diff(q_i, m'))$ *of some target program* m' *(where the latter means that no stronger predicate exists in* $diff(q_i, m')$*).*

In other words, a locally strongest user will always make progress using the most effective (i.e., strongest) predicates available. This means that, for instance when using GIM predicates [26], given a choice between two sequence exclusion predicates `exclude(`*drop*`)` and `exclude(`*drop* · *take*`)`, if they are both relevant, the user will

select the one making more impact – which is the sensible choice, as excluding the subsequence when the individual function is undesirable could cause it to appear again.

We notice that in case the sets of predicates in question have an infinitely decreasing (i.e., infinitely getting stronger) sequence of predicates, this restriction on the user is at odds with correctness: no predicate from the infinite decreasing sequence will be represented in its base set, which means the user may have a *correct* predicate available to them from $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m')$ but no action in the union on the base sets.

To counteract this, we would like to make sure every chain of predicates would have a strongest element to add to the base set. We therefore add a requirement for $\bigcup_{m' \in M^*} \beta(m')$ to be a well-founded order: we recall that if X is a wfo, it satisfies the strictly descending chain condition in definition 13(c) (but not necessarily the incomparable chain condition). The following lemma shows that if $\bigcup_{m' \in M^*} \beta(m')$ is a wfo, then a correct user that is able to make strong progress can also be locally strongest, i.e., it will never get stuck due to inability to find a "strongest" predicate.

Lemma 3. *Let $p \preceq p' \iff p \Rightarrow_M p'$. If $\preceq$ is a wfo over $\bigcup_{m' \in M^*} \beta(m')$, then whenever $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m') \neq \emptyset$, we have that $\bigcup_{m' \in M^*} \mathit{Base}(\mathit{diff}(q_i, m')) \neq \emptyset$ as well.*

Proof. First note that if $\preceq$ is a wfo over $\bigcup_{m' \in M^*} \beta(m')$, then it is also a wfo over $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m')$ for any $q_i \notin M^*$. This is immediate from the property that $\mathit{diff}(q_i, m') \subseteq \beta(m')$ and hence $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m') \subseteq \bigcup_{m' \in M^*} \beta(m')$. Since $\bigcup_{m' \in M^*} \mathit{diff}(q_i, m')$ is nonempty, well foundedness ensures that its base set is also nonempty, and hence also $\bigcup_{m' \in M^*} \mathit{Base}(\mathit{diff}(q_i, m')) \neq \emptyset$. □

We can now formalize our termination result for a locally strongest user. We start with the simpler case where M^* is a singleton set, and then extend it to the general case.

Theorem 2. *If $\bigcup_{m' \in M^*} \beta(m')$ is a wfo, $\bigcup_{m' \in M^*} \mathit{Base}(\beta(m'))$ is finite and the user is locally strongest, then any synthesis session that makes strong progress will converge in a finite number of steps.*

Notice that when using $\Rightarrow_M$ as an order relation, the requirement of finiteness of $\bigcup_{m' \in M^*} \mathit{Base}(\beta(m'))$ is similar to a wqo's finite basis requirement (Higman [13]). However, this requirement is only applied to $\beta(m')$ for $m' \in M^*$, not to all sets, and does not require an upwards-closed set. Also notice that if $\bigcup_{m' \in M^*} \beta(m')$ was a wqo, as required from theorem 1, this would already be true because of the finite basis property.

Proof. First we show that $\mathit{Base}(\mathit{diff}(q_i, m^*)) \subseteq \mathit{Base}(\beta(m^*))$ for every $m^* \in M^*$ and $q_i \in M$. Let us assume, by way of contradiction, that there exists a predicate $p \in \mathit{Base}(\mathit{diff}(q_i, m^*)), p \notin \mathit{Base}(\beta(m^*))$. We know that $p \in \beta(m^*)$, since $\mathit{diff}(q_i, m^*) \subseteq \beta(m^*)$, so for p to not be in $\mathit{Base}(\beta(m^*))$ there must be $p' \in \mathit{Base}(\beta(m^*))$ s.t. $p' \Rightarrow_M p$. p' is not in $\mathit{diff}(q_i, m^*)$, or it would also be in $\mathit{Base}(\mathit{diff}(q_i, m^*))$ instead of p, which means that $q_i \models p'$. However, since $q_i \not\models p$ and $p' \Rightarrow_M p$, we have reached a contradiction. This trivially implies that $\bigcup_{m' \in M^*} \mathit{Base}(\mathit{diff}(q_i, m')) \subseteq \bigcup_{m' \in M^*} \mathit{Base}(\beta(m'))$, and hence finiteness of $\bigcup_{m' \in M^*} \mathit{Base}(\beta(m'))$ ensures that $\bigcup_{m' \in M^*} \mathit{Base}(\mathit{diff}(q_i, m'))$ is finite as well.

Next we see that since $\bigcup_{m' \in M^*} Base(\mathit{diff}(q_i, m'))$ is finite, then if the user makes strong progress by selecting a predicate from $\bigcup_{m' \in M^*} Base(\mathit{diff}(q_i, m'))$ in each iteration, the session will always converge in at most $n \leq |\bigcup_{m' \in M^*} Base(\mathit{diff}(q_i, m'))|$ iterations when one of the following will occur:

- $\gamma(S_n) \subseteq M^*$ (as will be seen later in definition 16, $S_n = B \in \mathcal{B}$), and the session has converged successfully, or
- $\gamma(S_n) = \emptyset$, which means $q_{n+1} = \bot$, or the session has converged unsuccessfully.

The first option is a successful convergence. The second option, in which the session fails to converge successfully, is possible for two reasons. First, because our requirement for the user is not to select only from $\bigcup_{m' \in M^*} Base(\mathit{diff}(q_i, m'))$, and other correct user actions may still lead to a contradiction. Second, throughout the session, the user may select predicates from $Base(\mathit{diff}(q_i, m')) \subseteq \beta(m')$ of a different m', and these predicates may contradict. The latter is no longer a possibility if M^* is a singleton set. □

Example 5. Let us assume a singleton $M^* = \{m^*\}$, a domain of functional programs over a vocabulary $\mathcal{V}$ and a set of syntactic predicates $\mathcal{P} = \{include(seq), exclude(seq)\}$ predicates over all continuous sequences of methods $seq = f_1 \cdot f_2 \cdots f_n \in \mathcal{V}$.

We can see immediately that $\mathcal{P}$ itself is not a wfo: for every sequence used by $include$, there is a stronger predicate which includes a subsuming sequence. However, a specific target program m^*, and its description $\beta(m^*)$, is a different matter. While $exclude$ sequences can longer than the length of m^* as long as we wish and will still appear in $\beta(m^*)$, $include$ sequences that are longer than m^* will rule out m^*. This means that the chain of $include$ predicates in $\beta(m^*)$ is finite, and so $\beta(m^*)$ has a well-founded ordering.

6 Successful Convergence and Backtracking

In this section we characterize the cases where a synthesis session may converge successfully, in the sense that the user *has a path* that leads to successful convergence. We then examine situations in which a synthesis session trying to achieve a realizable target program goes awry and fails to converge successfully. The expected user behavior in these cases is to backtrack — to remove some of the provided specification or to cancel recent steps. We show that the point of realization that backtracking is needed is in many cases farther along the session than the point which necessitates backtracking. We explore the amount of sufficient backtracking, and show that it may be of any length.

Recall that a user's intention is *realizable* if $M^* \neq \emptyset$ (see Section 3). We observe that this is a necessary condition but in general not sufficient, and successful convergence requires a stronger notion of realizability. To formalize this notion, we need the following definition:

Definition 16 (Core set). *We say that a set $B \subseteq \mathcal{P}$ is a* complete specification *if* $\emptyset \neq \gamma(B) \subseteq M^*$. *We define the* core set *of the synthesis problem as the set of all* finite *specifications,* $\mathcal{B} = \{B \subseteq \mathcal{P} \mid \emptyset \neq \gamma(B) \subseteq M^* \wedge |B| \in \mathbb{N}\}$.

If there exists no $B \in \mathcal{B}$ such that $\emptyset \neq \gamma(B) \subseteq M^*$, then there is no finite underapproximation of the target space in the abstract domain defined by $\mathcal{P}$. In this situation,

every synthesis will always fail, even if the specification is technically realizable. Based on this observation, we define a stronger notion of realizability:

Definition 17 ($\mathcal{P}$-realizability). *We say that M^* is* $\mathcal{P}$-realizable *if $\mathcal{B} \neq \emptyset$.*

Indeed, $\mathcal{P}$-realizability is a necessary condition for successful convergence. For example example 3 describes the case in which the available predicates are syntactic predicates on a single function. If all programs in M that implement the user's intention are of length 2 or more, then there may not be an underapproximation of M^*. Likewise, when working with examples it may take infinitely many examples to differentiate between two programs (as shown in section 5), which means that the space described by any finite number of examples will still contain some program outside of M^*.

Even with $\mathcal{P}$-realizability, the user's steps may lead to a point where successful convergence is no longer possible. Next we generalize the above condition to refer to *any* point along the session. Furthermore, we show that the general condition is not only necessary but also sufficient for successful convergence (i.e., the user has a possible path to it). In order to provide the general condition we first define a property of the synthesizer's state that captures situations where successful convergence is out of reach.

Definition 18 (Inevitable failure point). *Let $\mathcal{S}$ be a session. The state S_i is called an* inevitable failure point *if $\forall B \in \mathcal{B}.\ \gamma(S_i) \cap \gamma(B) = \emptyset$.*

In particular, if $\gamma(S_i) \cap M^* = \emptyset$, then S_i is a point of inevitable failure. However, in general, this may not be the case — valid programs may exist even at an inevitable failure point (such programs are not contained in any $B \in \mathcal{B}$).

We note that the condition of an inevitable failure point can be equivalently defined as $\forall B \in \mathcal{B}.\ \gamma(S_i) \not\supseteq \gamma(B)$. Clearly, an empty intersection of $\gamma(S_i)$ with (the nonempty) $\gamma(B)$ implies that $\gamma(S_i)$ is not a superset of $\gamma(B)$. For the other direction, if there exists B such that $\gamma(S_i) \cap \gamma(B) \neq \emptyset$, then by taking the finite set $B' = S_i \cup B$ we get $\gamma(B') = \gamma(S_i) \cap \gamma(B) \subseteq \gamma(S_i)$. Moreover, $\gamma(B')$ is nonempty and included in $\gamma(B) \subseteq M^*$, hence $B' \in \mathcal{B}$.

Theorem 3 (Successful convergence). *Let $\mathcal{S}$ be the prefix of length n of a synthesis session. Then the following conditions are equivalent:*

1. *S_{n-1} is not an inevitable failure point,*
2. *there exists a session $\mathcal{S}'$ that extends $\mathcal{S}$ and converges successfully.*

Proof. The proof uses the equivalent formulation of inevitable failure point.

$2 \Rightarrow 1$ If $\mathcal{S}'$ converges successfully at step m, we select its final state S_{m-1} to be B. Because of the successful convergence, $\emptyset \neq \gamma(S_{m-1}) \subseteq M^*$, and since $S_{m-1} \sqsubseteq S_{n-1}$, then $\gamma(S_{m-1}) \subseteq \gamma(S_{n-1})$ (Galois connection).

$1 \Rightarrow 2$ Since S_{n-1} is not an inevitable failure point, there exists some B such that $\gamma(S_{n-1}) \supseteq \gamma(B)$. Since B is finite, the user can answer with $A_n = B$. Adding the step A_n leads to successful convergence: $S'_n = S_{n-1} \sqcap A_n = S_{n-1} \sqcap B$, so $\gamma(S'_n) = \gamma(S_{n-1}) \cap \gamma(B) = \gamma(B)$.

We note that unless $q_n \in M^*$ (in which case the prefix $\mathcal{S}$ is complete), the extension $\mathcal{S}'$ of $\mathcal{S}$ constructed from the non-inevitable failure point by selecting $A_n = B$ constitutes both weak and strong progress. The reason is that for $q_n \notin M^*$, $q_n \not\models B$, which makes this step a strong progress step, and, since some program has been eliminated, also a weak progress step.

Recall that convergence considers a worst-case synthesizer, which only returns a program from M^* when $\gamma(S_i) \subseteq M^*$. Theorem 3 implies that for such a synthesizer, if a synthesis session reaches an inevitable failure point, the session can either be infinite or end with $q_n = \bot$. This means that backtracking is necessary. However, in the worst case the failure may become observable to the user only when (if) the session terminates with $q_n = \bot$. A more sophisticated user may realize this earlier, at the first inevitable failure point where $\gamma(S_i) \cap M^* = \emptyset$. We refer to this point as the *first infeasible point*, and to the prior point as the *last feasible point*. We note that these points are only observable if $M^* = U^*$ (or if the user is aware of M^*).

6.1 Unbounded Backtracking

We now consider the amount of steps that have to be traced back from the point where $q_n = \bot$ (i.e., the session terminates with failure) or from the point where $\gamma(S_i) \cap M^* = \emptyset$ (i.e., the first infeasible point in the session) to recover a synthesizer state from which there is a suffix that leads to successful convergence. We argue that there is no bound on the number of steps that we need to backtrack; this is demonstrated via the following scenario.

Consider a syntesizer where M is all the programs in a language generated by `if` expressions, equality (==), all list constants over integers (e.g. `[]`, `[1, 2, 3]` etc.), recursive call f, the input variable `i`, and the library functions `cons`, `max`, `remove`, `sort`, and `reverse`.

The predicate set $\mathcal{P}$ contains all input-output examples (ι, ω), and syntactic exclusion of a single element, that is "exclude e" for $e \in \{$`if`, ==, `cons`, $\cdots\}$.

The user wants to sort a list of integers in descending order. The following table shows a possible interactive session with the synthesizer.

i	A_{i-1}	q_i
1	$([], [])$ $([1, 2], [2, 1])$	`reverse(i)`
2	exclude `reverse`	**`if`** `(i == [1, 2]) [2, 1]` **`else`** `i`
3	$([1, 3], [3, 1])$	**`if`** `(i == [1, 2]) [2, 1]` **`else if`** `(i == [1, 3]) [3, 1]` **`else`** `i`
$\vdots$		
n	exclude ==	$\bot$

The first two examples lead the synthesizer to generate a simple list reversal program. The user is not interested in this program, and disqualifies it by excluding `reverse`. The synthesizer then, quite unfortunately, takes the path of over-fitting the example set via branching using the `if` construct with equality conditions. The user keeps providing examples, but is handed an ever-growing chain of programs. After n such steps, the

user chooses to block the synthesizer from over-fitting to particular inputs by excluding the equality operator, at which point the synthesizer can no longer find a program in M that satisfies S_{n-1}, and $Select$ returns $\bot$.

Core set The core set $\mathcal{B}$ for this instance is the set of all finite sets of predicates containing no contradiction and (at least)

- One of $\{$exclude `if`, exclude ==$\}$
- Two examples $\{(\iota_1, \omega_1), (\iota_2, \omega_2)\}$ with $\iota_{1,2}$ two lists such that $\left|\langle x \in \iota_1 \mid x > \text{head}(\iota_1)\rangle\right| > |\iota_2|$, and $\omega_{1,2}$ their corresponding descending sorts.

To see why this is the core set, first note that the exclusion of either `if` or ==, rules out conditionals as well as any form of recursion (since any recursive call will then be infinite). Including two input examples with the specified property rules out programs that use `remove` to reorder the elements. [3] Moreover, when excluding neither `if` nor ==, no number of examples is sufficient to make a complete specification since `switch`-like over-fitting is always a valid solution.

Inevitable failure point In this example, an inevitable failure point occurs after the second step. The reason being, that any $m \in \gamma(B)$ must use `reverse`, since any non-recursive program without it can correctly order only a fixed number of elements from the input. $\{$exclude `reverse`$\}$ disallows that, leading to $\gamma(B) \cap \gamma(S_1) = \emptyset$.

It is possible for a correct user to reach this state, since the user expects the program `sortBy(i, neg)`, which is a valid program ($\in U$) — but this program is beyond the synthesizer's search space ($\notin M$).

First infeasible point It should also be noted that that after the second step, $\gamma(S_1) \cap M^* \neq \emptyset$, since `if (i==[]) [] else cons(max(i), f(remove(i, max(i)))` (also known as max-sort) is a realization of the goal. So S_1 is still a feasible point, and so are $S_{2..(n-2)}$ — since the examples consist of valid descending sorts, hence max-sort $\models A_{2..(n-2)}$. Max-sort is only discarded at A_{n-1}, by the exclusion of ==, and since `reverse` has already been excluded, `reverse(sort(i))` or any other composition of `sort` and `reverse` cannot be generated. Now, $\gamma(S_{n-1}) \cap M^* = \emptyset$, making iteration n the first infeasible point. It so happens that the three examples shown are enough to make $\gamma(S_{n-1})$ empty, so the synthesizer returns $\bot$.

The last, important thing is that we can construct the session with an arbitrarily large n, such that the inevitable failure point ($i = 2$) is any number of steps away from the last feasible point ($i = n - 1$), and also from the actual failure with $\bot$ ($i = n$). It means that any bounded backtracking is insufficient for recovering the session in this case.

Theorem 4. *For any given $k \in \mathbb{N}$, there exist:*

1. *a session $\mathcal{S}$ of length $k + i$ where S_i is an inevitable failure point and $q_{k+i} = \bot$.*
2. *a session $\mathcal{S}$ where S_i is an inevitable failure point and S_{k+i} is the first infeasible point.*

[3] The number of `remove`s has to be at least $\left|\langle x \in \iota_1 \mid x > \text{head}(\iota_1)\rangle\right|$, but at most $|\iota_2|$, which is not possible without branches.

Proof. Using the construction described above, having $i = 1$ and either $n = k + 1$ (for 1) or $n = k + 2$ (for 2). Notice that in this scenario, the nth iteration exhibits both a first infeasible point and failure with $\bot$.

7 Discussion

In this section we discuss the implication of some of the conditions posed in definitions and theorems in the previous sections.

7.1 Progress models

Progress of the synthesizer is important not only for making sure the session will converge, but also as a tool for the user to understand their status in the synthesis session.

Synthesizers that do not actively define themselves as iterative have no way of enforcing progress, of course, but if the implementation of $Select$ is order-dependent, then the user can tell whether their feedback has moved the session along. This is tricky when considering weak progress—$Select$ might stop at the same program even though other programs have been eliminated from the space. This, of course, is the danger of weak progress. One of the reasons it would be helpful for synthesizers to start considering themselves as interactive is so they can provide this feedback to the user, and limit frustration and confusion.

We have already seen an example of a synthesizer that enforces very strict strong progress in FlashFill, and FlashExtract [20] and BlinkFill [30] follow the same workflow. GIM [26] puts forth a set of predicates that allow the user to provide positive feedback on the program, which means that even if strong progress is to be enforced, it must be enforced at the more relaxed level described in definition 10, allowing predicates that hold for the current program along with those that rule it out. In an enumerating synthesizer that unifies sub-programs based on observational equivalence, such as [24], weak progress may be sufficient: a change in the search space could change the equivalence classes created while enumerating, leading to a different result from $Select$ even though the current program was not eliminated. This could also aid a realistic user who might not be completely certain whether a program is in M^*.

When designing a new synthesizer, there are pros and cons to each of the progress models. Strong progress, paired with a $Select$ that will return the same program again and again, will reduce user frustration. Weak progress has been shown [26] to help an uncertain user reach a better program. However, the feasibility of enforcing the progress model is itself an issue: strong progress is easy to test, as it only requires for the user answer to rule out the current program. Weak progress, as seen in lemma 1, requires the ability to check implication of the predicates over the current domain of programs. This, even for simple predicates, may be difficult.

There is also the possibility of not enforcing progress at all. It can be easily seen that termination, as proved in section 5, is not impeded if the user provides finitely many answers that do not make progress along with those that do. (This also applies to finitely many steps made by predicates for which termination is not guaranteed). However, we believe forcing progress is a way to keep the user on track.

7.2 Realizability gap

One of the problems a synthesizer can suffer from is a gap between the expectations of the user and the ability of the synthesizer. Often, this is expressed by the fact that $M^* \subset U^*$, as in the example in section 6.1. In such a case, a user can repeatedly backtrack and try new predicates, and still fail because they may not even be able to pinpoint the first infeasible point of a session, let alone the initial point of inevitable failure of their session.

Unfortunately, there is not much that can be done about this, especially since limitations on the expressibility of M have been previously shown to be important for both termination [22] and for heuristically arriving at the user's intentions faster [20]. All that remains for the synthesizer to do is to better communicate the limitations of M.

7.3 Sharing more with the user

One of the design tenets behind [26] is to enrich the interaction model with the user and to include more information about the program. Another way in which the interaction can be made more informative is by communicating more information about the state of the synthesizer. Section 7.1 suggesting an indication of whether, and what level, of progress has been made is an example of this.

Similarly, the synthesizer can communicate additional data about M and $\mathcal{P}$. Showing the user a visualization of the remaining search space may help with problems such as the realizability gap or to identify points of failure faster. Suggesting to the user stronger predicates they may wish to use in their answer might help the process terminate faster.

8 Related Work

Programming by Example In PBE the interaction between user and synthesizer for demonstrating the desired behavior is restricted to examples, both in initial specifications and any refinement. FlashFill [10, 29] is a PBE tool for automating transformations on an Excel data set, and is included in Microsoft Excel. Its implementation is based on the theory of Version-Space Algebra [19]. FlashFill is iterative by design, accepting a (strong progress) update to its specification if the resulting program is not satisfactory. The FlashMeta family of synthesizers [20, 29, 30] follow this same trend.

Counterexample-guided inductive synthesis CEGIS is a synthesis framework that has been formalized in [31] and [21]. It is implemented in tools such as Sketch [32, 33], which allows the user to restrict the search space via structural elements (e.g. conditions or loops) containing holes to be synthesized. Sketching is a way to leverage a programmer's knowledge of expected syntactic elements, and when used in conjunction with restrictions on the syntax [1] can allow very intricate synthesis. Sketch exhibits two forms of iterative processes: the first one is an internal loop that involves a solver and a verifier, where the solver attempts to fill the holes in the sketch and the verifier provides a stream of input-output examples until the result passes validation; and the second, external one involves the human user and the tool, where the user may not like the generated program or the tool rejects the sketch because it is unsatisfiable. The internal loop is example-driven, with the verifier taking the place of the user. The external one is non-monotonic, as the user can remove assertions from the specification

or change the syntactic class of the program entirely. The only monotonic changes are (i) adding an assertion, (ii) removing an assumption, and (iii) replacing a numeric hole with a constant.

Type-directed synthesis In type-directed synthesis tools such as [9, 12, 27], the specification is provided entirely by types. These tools tend to not use an iterative model, as refining the specification is not trivial. Synquid [28] is a type-directed synthesis tool that uses refinement types, which encode constraints on the solution program to be imposed on the candidate space. Refinement types have rich semantics and a definition of subtyping based on logical implication. The user can add syntactic structure (roughly, the top of the tree) to help the synthesizer, and can also strengthen the return type of the program (by replacing it with a subtype) or loosen the precondition for the types of the arguments (by replacing them with a supertype). These are all monotonic progression steps, but the user can also change a type to any other type or change the number of inputs to the program, which are not monotonic. Tools that combine type-directed synthesis with examples [6, 7, 24] make for a more iterative model, as adding examples is always monotonic.

Formal models of synthesis procedures Models of families of synthesizers exist for enumerative, syntax-based synthesizers [1], VSA-based synthesizers [29], and oracle-driven synthesizers via inductive learning [14]. These all describe a single-iteration interaction with the user (though [14], which describes the counterexample-driven model as well, does describe iterative behavior with the oracle). Two recent works describe an iterative model of interactive synthesis. One [21] focuses on the synthesizer-driven model of interactive synthesis: the synthesizer asking the user about differentiating examples, and turning the answer back into constraints on the search space. This model is somewhat specialized for VSA-based synthesizers and is an interactive expansion of [29]. The work of Loding et al. [22] which is intended mostly to describe the internal iteration of a CEGIS synthesizer, is also suited to a user-driven model of interactive synthesis, as is the one presented in this paper. The model is based in machine learning terminology, with a teacher-learner model exploring a hypothesis space (i.e., a space of programs or other classifiers), and use a sample space containing input-output examples and no additional forms of feedback. Finally, they offer a weaker termination result, showing the existence of a terminating learner (user) hinging on an ordering of the hypothesis space.

Acknowledgements

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME.

Bibliography

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015.

[2] T. Anton. Xpath-wrapper induction by generalizing tree traversal patterns. In *Lernen, Wissensentdeckung und Adaptivitt (LWA) 2005, GI Workshops, Saarbrcken*, pages 126–133, 2005.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[4] D. Drachsler-Cohen, S. Shoham, and E. Yahav. Synthesis with abstract examples. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 254–278, 07 2017.

[5] 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. *Science of Computer Programming*, 69(1):35–45, 2007.

[6] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2017*, 2017.

[7] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[9] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 653–663. ACM, 2014.

[10] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[11] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 8–14. IEEE, 2012.

[12] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[13] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336, 1952.

[14] S. Jha and S. A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, Feb 2017.

[15] D. E. Knuth. Computer programming as an art. In *ACM Turing award lectures*, page 1974. ACM, 2007.

[16] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.

[17] J. Landauer and M. Hirakawa. Visual awk: A model for text processing by demonstration. In *vl*, pages 267–274, 1995.

[18] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Learning repetitive text-editing procedures with smartedit. *Your Wish Is My Command: Giving Users the Power to Instruct Their Software*, pages 209–226, 2001.

[19] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1):111–156, 2003.

[20] V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In M. F. P. O'Boyle and K. Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.

[21] V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani. Interactive program synthesis, 2017.

[22] C. Löding, P. Madhusudan, and D. Neider. Abstract learning frameworks for synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–185. Springer, 2016.

[23] A. Omari, S. Shoham, and E. Yahav. Cross-supervised synthesis of web-crawlers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 368–379, 2016.

[24] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.

[25] H. Peleg, S. Shoham, and E. Yahav. D3: Data-driven disjunctive abstraction. In *Verification, Model Checking, and Abstract Interpretation*, pages 185–205. Springer, 2016.

[26] H. Peleg, S. Shoham, and E. Yahav. Programming not only by example, 2017.

[27] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Notices*, volume 47, pages 275–286. ACM, 2012.

[28] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 522–538. ACM, 2016.

[29] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[30] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. September 2016.

[31] A. Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[32] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *ACM SIGPLAN Notices*, volume 43, pages 136–148. ACM, 2008.

[33] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review*, 40(5):404–415, 2006.

[34] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM, 2017.

[35] I. H. Witten and D. Mo. Tels: Learning text editing tasks from examples. In *Watch what I do*, pages 183–203. MIT Press, 1993.

[36] S. Wu, J. Liu, and J. Fan. Automatic web content extraction by combination of learning and grouping. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1264–1274. ACM, 2015.

Generating Tests by Example

Hila Peleg, Dan Rasin, Eran Yahav

Technion {hilap,danrasin,yahave}@cs.technion.ac.il

Abstract. Property-based testing is a technique combining parametric tests with value generators, to create an efficient and maintainable way to test general specifications. To test the program, property-based testing randomly generates a large number of inputs defined by the generator to check whether the test-assertions hold.
We present a novel framework that synthesizes property-based tests from existing unit tests. Projects often have a suite of unit tests that have been collected over time, some of them checking specific and subtle cases. Our approach leverages existing unit tests to learn property-based tests that can be used to increase value coverage by orders of magnitude. Further, we show that our approach: (i) preserves the subtleties of the original test suite; and (ii) produces properties that cover a greater range of inputs than those in the example set.
The main idea is to use abstractions to over-approximate the concrete values of tests with similar structure. These abstractions are then used to produce appropriate value generators that can drive the synthesized property-based test.
We present JARVIS, a tool that synthesizes property-based tests from unit tests, while preserving the subtleties of the original unit tests. We evaluate JARVIS on tests from Apache projects, and show that it preserves these interesting tests while increasing value coverage by orders of magnitude.

1 Introduction

Parametric unit-tests [52, 51, 45, 53, 55] are a well-known approach for increasing coverage and thus increasing confidence in the correctness of a test artifact. Parametric unit tests (PUTs) are also a common target of automatic test generation [22, 7] and unit test generalization [23, 50, 10]. A parametric unit test consists of a *test body* defining the parametric code to execute, and a set of *assumptions* that define the requirements from input values.

Parametric unit tests can either be symbolically executed or *instantiated*, which is the process of turning them back into unit tests [52, 51]. One way to instantiate PUTs is to provide them with concrete values based on whitebox knowledge of the program under test [53, 58]. Another way is to provide a value generator for the parameters, usually hand-crafted by an expert, which generates appropriate values on demand. This type of test is called a *property-based test* (PBT) [19, 11, 16, 30].

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 406–429, 2018.
https://doi.org/10.1007/978-3-319-73721-8_19

The paradigm of property-based testing [13, 4, 1, 43] defines the desired behavior of a program using assertions on large classes of inputs ("property"). To test the program, property-based testing generates inputs satisfying the precondition to check whether the assertion holds. Property-based testing is known to be very effective in checking the general behavior of the code under test, rather than just on a few inputs describing the behavior. It does this by describing the behavior as assertions over classes of input, generating random inputs from that class to check the assertion against. This has the advantages of increasing both instruction coverage and value coverage, and exposing bugs which may be hidden behind the selection of specific representative test cases.

In this paper we present a technique for automatic synthesis of PBTs—parametric unit tests, together with an appropriate value generator—from repetitive unit tests.

The value generators synthesized by our approach follow relationships captured by an abstract representation to explore values within the test's input assumptions. In contrast to the assumptions of parametric unit tests which require a separate enumeration technique (e.g., based on whitebox guidance), abstraction-based generators contain nothing but the definition of the desired input space, and so can be sampled directly and repeatedly to provide a large number of additional values that satisfy the required assumptions.

Our approach generalizes existing unit tests by finding tests with a similar structure such that their concrete values can be over-approximated using an abstract domain. This allows us to use the executed code from the original test, as well as the oracle (assertion) of the test, and execute them with new concrete values. Our approach learns from both positive and negative test-cases (i.e., tests expected to succeed and fail, resp.), enabling a more precise generalization of tests. Specifically, it finds an over-approximation for the positive examples, while excluding any negative examples, and vice versa. In addition, our generalized tests preserve constraints inherent in concrete unit tests, such as types and equalities, which allow us to address the subtle nuances tested by them.

Challenges To achieve our goal, we have to address the following challenges:

– Identify which tests, along with their oracles, should be generalized together to obtain parametric tests.
– Generalize matching tests to find an over-approximation that represents all positive examples but none of the negative ones. This will allow us to synthesize value generators that match the generalized tests.

Existing Techniques Thummalapenta et al. [50] conducted an empirical study analyzing the cost and benefits of manually generalizing unit tests, and have shown that the human effort pays off in increased coverage and newly detected bugs. Shamshiri et al. [46] conducted a test of state-of-the-art test-generators and concluded that they do not create tests as meaningful as human-written tests, leading to the conclusion that basing generalization on existing tests will lead to better results. Fraser and Zeller [23] create tests with pre- and postconditions on parameters, but do so by assuming a baseline version of the program and, in

practice, incorporate its bugs into the tests. Francisco et al. [21] created PBTs for web services, but did so from a semantic description that had to be manually written for each web service. Loscher and Sagonas [34] improve upon PBTs with guided value generation, rather than simple random sampling.

Our Approach The main idea is to leverage the repetitive nature of existing unit tests to automatically synthesize parametric tests and generators. Technically, we define a partial order on the set of tests, that captures the generality of the test data. This order allows our technique to use the same unit test as an example for several different PBTs, capturing different subtleties, and at the same time staves off over-unification of example sets that would yield meaningful results individually, but a non-informative generalization together. We use safe generalization [42] to separate positive and negative examples.

Dividing Tests Provided with individual example unit-tests to be used as a training set, we aim to divide them into sets to be abstracted, in order to create the smallest number of abstractions that are still meaningful, and can still be sampled. We then aim to determine how many value generators are to be created for each such abstracted region of the parameter space. The goal of the division is to create a set of value generators for the property-based tests that will be generated such that each abstracted region can over-approximate the maximal number of examples, and different value generators are created over the same region preserve the testing nuances seen in the original tests. The motivation is that a generator for a PBT must contain the constraints of the subtle cases that were selected by the programmer, to guarantee that these cases are covered in a non-negligible probability when the PBT is executed.

To support this goal, we define a partial order of generality between PBTs. This allows us to create a value generator for each testing nuance, and do so on the maximal number of examples that are compatible with this subtlety.

Safe Generalization of Tests Given a set of compatible positive tests (expected to succeed) and negative tests (expected to fail), we wish to generalize them into a region that a PBT's value generator can sample. To that end, we use an abstraction method for separating positive and negative examples, called *Safe Generalization* [42].

Implementation We present JARVIS (Junit Abstracted for Robust Validation In Scalacheck), a tool that extracts repetitive tests from unit test suites, determines their place in the partial order, and synthesizes from them PBTs that generate inputs based on preserved properties. We test JARVIS on unit tests from Apache projects. We also show that sampling the abstracted over-approximations increases value coverage[8, 29] of the exercised code while not losing instruction coverage. In addition, we demonstrate JARVIS's ability to discover historical bugs when run on test suites in the previous versions.

Main Contributions The contributions of this paper are:

- An inclusion relation between parameterized tests that allows the sharing of examples between different abstracted generators without hindering the ability to abstract.

- A technique that generalizes test values from individual unit tests into value generators for a PBT using safe generalization (separating positive and negative examples).
- A tool, JARVIS, that automatically synthesizes parametric tests, oracles and abstraction-based generators from unit tests, while preserving the subtle cases that are captured in these tests.

2 Overview

Unit tests are an integral part of the software development process. They are used to test small components of large software systems independently. Such components can typically receive many possible inputs, and in order to cover their different behaviors, a component is often run using the same test code with several different input values. In practice this leads to repetitive test code to exercise the same unit under test again and again. An initial study of the repetitiveness in the test suites of five large Apache projects (Commons-Math, Commons-Codec, Collections, Sling and Spark core) showed that of $13,359$ total tests, 40% are not unique test scenarios, and 17% are repetitive by being written as an assertion called inside a loop. In some test files, *all* test code is non-unique either by virtue of repetition or loops. This means that repetition of individual tests is not only present but frequent.

However, these tests still use the same values every time the test suite is run. Running identical code with other possible values may reveal a bug, and new bugs may be introduced that will not be tested because of the test values are constant. In fact, tracing through the history of the testing code shows us many such cases: identical tests with a small change of constant values that were later added to represent a bug that has been discovered, and often has been in the code for a full version or more.

We set out to take repetitive test suites and synthesize from them testing properties for property-based testing. Once we have in our possession a parameterized test with an assertion to test its postcondition, as well as a set of values for the parameters labeled for expected success or failure of the test, we can use previous work [47, 49, 33, 20, 17] to learn a precondition on the data and convert it to a data generator for a PBT.

However, dividing test traces into compatible sets is not trivial. Tests may seem to be representing the same case but in their over-unification harm the abstraction. In addition there may be sets of tests that represent an interesting test case, such as equal parameters or a subtype being used, which should be preserved when sampling.

This paper addresses the following problems:

1. Finding individual tests that can be generalized together ("compatible");
2. Generalizing the tests into a property-based test that would cover a superset of the original tests; and
3. Creating abstraction-based value generators that will sample the abstraction while preserving testing nuances.

```
1  Assert.assertTrue(Precision.equals(153.0000, 153.0000, .0625));
2  Assert.assertTrue(Precision.equals(153.0000, 153.0625, .0625));
3  Assert.assertTrue(Precision.equals(152.9375, 153.0000, .0625));
4  Assert.assertFalse(Precision.equals(153.0000, 153.0625, .0624));
5  Assert.assertFalse(Precision.equals(152.9374, 153.0000, .0625));
```

Fig. 1. Several unit tests from the test suite of the Apache commons-math project, using the JUnit testing framework.

To solve 1 we define the notion of tests that are compatible—that test the same thing, and so have the same notion of correctness behind the examples. To solve 2, we use the notion of Safe Generalization in order to find an abstraction that will separate completely the example test cases that are expected to succeed from those that are expected to fail. Finally, to solve 3, we sample these abstractions in a constrained manner dictated by the original tests.

We demonstrate these steps on a real-world example taken from the Apache Commons-Math test suite.

The code segment in Fig. 1 depicts duplicate tests with different constant values in the class `PrecisionTest` in the Commons-Math project. We notice that the seemingly straightforward duplication is not exact duplication. For instance, the test in line 1 uses the same value twice, creating an equality constraint. In fact, in the larger file, there are *several* such tests, using different constants but repeating the value between the first and second parameter.

This means that there is an explicit intention to test the case where the two parameters are equal. Leaving this to chance while drawing reals would make getting two equal values highly unlikely, and the synthesized property would be skipping an intentional special test case if this is not performed. We therefore wish to generate as our output not one but two tests: one for the general case and one for the test with the equality constraint.

Parameterized tests Each test trace is turned into a *parameterized test*. In a parameterized test, constants are extracted and replaced by parameters of the same type. Parameter extraction takes into account constraints that exist in the concrete test, which means that if the same value appears more than once it will be extracted as the same parameter every time. For instance, `assertTrue(Precision.equals(153.0000, 153.0000, .0625)));` (line 1) will be parameterized to $pt_1 =$ `assert(Precision.equals(x, x, y));` with types $type(x) = type(y) = double$, and the parameter mapping of $\{x \mapsto 153.0, y \mapsto .0625, res \mapsto +\}$ is preserved, where res signifies the expected result of the assert. Similarly, lines $2-5$ will all be parameterized into $pt_2 =$ `assert(Precision.equals(x, y, z));` with $type(x) = type(y) = type(z) = double$, with four matching parameter mappings.

Grouping parameterized tests into scenarios Parameterized tests that test the same sequence of statements but for the different parameters are grouped together into *scenarios*. All parameterized tests in such a scenario would yield a property-based test that runs the same code, only with differently drawn values

for the parameters. In Fig. 1, both the parameterized tests pt_1 and pt_2 are testing `assert(Precision.equals(?,?,?))` and will be grouped into the same scenario.

All parameterized tests in the same scenario execute the same trace, or in other words test the same thing. A naive solution could use the parameterized data from all test traces belonging to a scenario, and simply perform the abstraction on them, generating a single property-based test for the entire scenario. However, because of the transition from constant values to randomly generated ones, information about the intent of the test is lost. E.g., if the parameterized test sends an integer to a double argument of a method, there is an intent for a number with no fractional part. If the parameterized test repeats a value throughout the test (e.g. between method arguments) there may be an intent for equality. In both cases, the chance of obtaining a value that fits the intention when drawing random values—e.g. from $\mathbb{R}^3$ in the case of Fig. 1—is slim at best.

A simple solution for this could be to keep the tests separated by the parameterized tests that contain them. This means all examples from tests that match `assert(Precision.equals(x,x,y))` with $type(x) = type(y) = double$ will be joined, separate from those that match `assert(Precision.equals(x,y,z))` with $type(x) = type(y) = type(z) = double$. This would generate an additional test forcing the equality of arguments, but would withhold from the unconstrained case with three parameters the additional data points that were separated out. Since both these parameterized tests call the same method, these data points contribute to the understanding of the method's general behavior, and this would cause the generalization of the second test to learn from fewer samples.

A hierarchy of tests A more realistic solution is to abstract as many examples as can be safely unified together, and sample each abstracted region separately later. To do this, we create a hierarchy of parameterized tests based on their parameters. For each parameterized test, we may also consider the data from all the tests below it in the hierarchy. When creating abstractions for the scenario, we consider the maxima of the hierarchy, along with all additional tests that have propagated up to them. This shares as many examples as possible, while preventing over-unification.

To do this, we define an inclusion relation between parameterized tests belonging to the same scenario, based on the sequence of all parameter uses in the test trace. In our example, pt_1 has the parameter sequence $x \cdot x \cdot y$ whereas pt_2 has the parameter sequence $x \cdot y \cdot z$.

We will say pt_1 is a *subtest* of pt_2 because (i) every parameter in place i in the sequence for pt_1 has an implicit conversion to the parameter type of the parameter in place i in the sequence of pt_2, and (ii) any equality constraint in the usage sequence of pt_2 (i.e. the parameter is repeated between places i and j) is also present in the sequence of pt_1. In this case, (i) holds trivially as the types are the same, and (ii) holds because the constraints in pt_1 are relaxed to no constraints in pt_2.

Section 4.2 details the $\sqsubseteq$ relation between two parameterized tests. Section 7 presents experimental data on the importance of using the hierarchical approach.

Abstracting the test data Now that the parameterized tests have been ordered and their concrete samples shared, we can abstract the values of the maxima of the $\sqsubseteq$ relation to a more general behavior. Earlier we parameterized the expected result of the trace with the assignment for res, indicating whether the concrete test should succeed when tested with the constants in the current parameter assignment. This can be used as a label for the parameter assignments as positive or negative examples of the more general property, which we wish to abstract. The examples comprising pt_2 yield the following two sets:

$$\begin{aligned} Positive = \{&(153.0000, 153.0000, .0625),\\ &(153.0000, 153.0625, .0625),\\ &(152.9375, 153.0000, .0625)\}\\ Negative = \{&(153.0000, 153.0625, .0624),\\ &(152.9374, 153.0000, .0625)\} \end{aligned}$$

We are interested in finding an abstraction for the $Positive$ and $Negative$ sets which explains the partition above, and enables us to generate many more positive and negative examples. It is vital that the abstraction will create a clear-cut separation between the positive and negative examples, in order to ensure that the values drawn will be a superset of the existing examples. This is also a reason that having a large example set is important: having more examples helps grow the abstraction, and having more counterexamples will limit the positive abstraction from covering portions of the input space that should be negative.

To do this, we use the notion of Safe Generalization, and abstract both the positive and negative samples simultaneously, checking that the abstraction of positive examples has not grown to cover negative examples and vice versa.

If there are several maxima in the relation that are being abstracted separately, we notice that the $Negative$ set for each of them contains negative examples for the scenario behavior. This means each $Positive$ set should be separated from *all* $Negative$ sets, and vice versa. These additional points to be used as counterexamples will improve the separation.

In our case, the abstraction describing the positive examples is $|x - y| \leq z$, and its negation for the negative examples. Section 5 formally defines Safe Generalization and details the use of JARVIS's template library.

In cases where the abstraction is performed on very few samples, there is a lot of room for error for any abstraction. In other programming by example tools such as [28, 31], the solution is to allow the user to mark the solution as incorrect and provide more examples. Section 5.1 discusses the reasons the abstraction may not be ideal and possible solutions.

Sampling the abstraction Once an abstraction is obtained for some set $PT = \{pt_1, pt_2, \ldots, pt_n\}$ of parameterized tests, we turn our attention to sampling the abstracted region, and to the preservation of testing nuances. Because we consider test cases written by the user a weighted sampling of the abstract behavior, we want to make sure we model the sampling of our PBTs in the same fashion.

```
1  val gen_double_1_pos = for(
2    y <- Arbitrary.arbitrary[Double].map(Math.abs);
3    x <- Arbitrary.arbitrary[Double];
4    z <- Gen.choose[Double](x - y, x + y)
5  ) yield (x,y,z)
6  forAll (gen_double_1_pos) {_ match {
7    case (d1: Double,d3: Double,d2: Double) =>
8      Precision.equals(d1, d2, d3)
9  }}
10 val gen_double_1_neg = for(
11   y <- Arbitrary.arbitrary[Double].map(Math.abs);
12   x <- Arbitrary.arbitrary[Double];
13   z <- Gen.oneOf(
14     Gen.choose[Double](Double.MinValue,x - y).suchThat(_ < x - y),
15     Gen.choose[Double](x + y,Double.MaxValue).suchThat(_ > x + y))
16 ) yield (x,y,z)
17 forAll (gen_double_1_neg) {_ match {
18   case (d1: Double,d3: Double,d2: Double) =>
19     !(Precision.equals(d1, d2, d3))
20
21 }}
```

Fig. 2. The ScalaCheck properties synthesized from the test traces shown in Fig. 1.

To do this, we generate an abstraction-based value generator for each $pt_i \in PT$, which will practice *constrained sampling*, i.e., draw values from the abstract region under the parameter constraints of the parameterized test. Section 6 details the way value generators are created over the abstract region.

Finally, we synthesize a PBT to includes each value generator. Fig. 2 shows the resulting properties both the positive and negative data abstractions applied to the concrete samples in $PT = \{pt_1, pt_2\}$, sampled according to pt_2.

Running this property will test the parameterized test on hundreds of values each time. This means that values matching the expected behavior but not covered by the concrete tests will now be tested. This can find bugs that are simply not tested for, and if the test property is added to the test suite, can help stave off bugs that will be added in future changes to the code. Section 8 shows a case study of a historical bug in Apache Commons-Math that was found by using JARVIS on the library's test suite in the version before the bug was corrected.

3 Preliminaries

In this section we introduce concepts used in this paper, including property-based testing and value-based coverage metrics.

Unit test A *unit test* consists of stand-alone code executed against a Unit Under Test (UUT), the result of which is tested against an oracle (an assertion) for correctness. In practice, the code exercising the UUT often targets a small unit, and the oracle is implemented by a set of assertions testing the state and output of the unit test code. Unit testing tools such as JUnit [2] and NUnit [3] provide an environment that can execute an entire *test suite* of unit tests.

Property-based test PBTs consist of test code and an oracle that are defined over parameterized classes of values. For that class of values, the PBT is phrased as a "forall" statement or axiom on the behavior of a component. This means PBTs mirror not a specific code path, but the specifications of the UUT. For example, a simple property on strings would specify that $\forall s_1, s_2, len(s_1 \cdot s_2) = len(s_1) + len(s_2)$.

A property-based test is comprised of two parts: the test body and oracle, which are the code operating on the UUT and the boolean statement which must hold, in this example concatenating and testing the length of strings; and the *generator*, which defines the class of inputs on which the PBT is defined, in this example any two non-null strings.

This is similar to the way *parameterized unit tests* [52] are defined. However, PUTs define the input class by assumptions on the parameters. This means that in order to run as tests in the test suite, PUTs need to be run through a solver or a symbolic execution of the UUT in order to be instantiated with values for the parameters, methods which are usually whitebox. The instantiated parameters are added to the test, which is then transformed into a conventional unit test. Barring a re-run of the solver, the values on which the resulting tests are run are constant.

In contrast, PBTs are intended for execution of the test body on a random sample of values that are drawn from the generator. The generator, rather than describing the input class as a boolean formula (i.e., the conjunction of all assumptions) that filters inputs, defines concretely a portion of the input space from which values can be drawn.

A test using the generator can be added as-is to a test suite using PBT frameworks such as QuickCheck [30, 13], PropEr [41], JSVerify [1] and ScalaCheck [4] that include an initial implementation for the building blocks of generators, such as ScalaCheck's `Gen.choose` used in Fig. 2.

It has been shown [50] that test parametrization is worthwhile in terms of the human effort it requires and the bugs that are detected. It can be extrapolated that PBTs, for which it is easier to draw a large set of test values, would be a worthwhile substitute.

4 Compatible Tests

4.1 From Test Trace to Parameterized Test

In this section, we formally describe how different unit test within a single test suite can be viewed as a repetition of the same test with different parameters. We then continue and formalize what we consider as subtle cases, or testing nuances, appearing is such a group of repetitive tests, and explain how our technique still preserves them.

The first step of our technique is to identify test traces in the original test suite. A *test trace* is a sequence of (not necessarily adjacent) statements ending with a single tested assertion, that can be executed sequentially.

For example, lines $3-4$ of Fig. 3 form the test trace `Interval interval = new Interval(2.3,5.7); assertEquals(3.4,interval.getSize());`. Each line in Fig. 1 forms its own test trace, e.g. `assertTrue(Precision.equals(153.0,153.0,.0625));` is formed by line 1.

To handle the many test traces in a library's test suite, we must group them into sets of tests that are compatible for a common abstraction. To this end, we first normalize them and create tests that do not use any specific constant values. This normal form is called a *parameterized test*. Technically, a parameterized test obtained from a test trace contains the same statements as in the test trace, where constant values are replaced by an uninterpreted parameter of the same type as the constant. Moreover, if the same constant appears multiple times in the test trace (at different locations), all occurrences are replaced by the same parameter. Finally, specific assertions such as `assertTrue` or `assertFalse` are replaced with a general `assert` command.

As seen in Section 2, the test trace in line 1 of Fig. 1 is parameterized as $pt_1 =$ `assert(Precision.equals(x, x, y));` with types $type(x) = type(y) = double$. Similarly, the test trace in lines $3-4$ of Fig. 3 is parameterized as `Interval interval = new Interval(x,y); assert(z==interval.getSize());` with $type(x) = type(y) = type(z) = double$.

Note that while a concrete test trace holds correctness information (i.e., the desired result of the assertion on a concrete execution of the trace), a parameterized test no longer encodes any such information. The expected result of the assertion is stripped along with the constant values, as it depends on them: the exact same parameterized test might be a positive test on one set of values and a negative test on another.

The relation between a parameterized test and a test trace from which it was originated, relies on the following definition:

Definition 1 (Parameter mapping). *A parameter mapping for a parameterized test is a function f that maps every parameter x to a constant $c = f(x)$ s.t. $type(x) = type(c)$. Additionally, f maps a new variable res to $\{+, -\}$.*

Essentially, a parameter mapping is a function that reproduces the original test trace from a parameterized test. The role of *res* in the definition above is to represent the type of the assertion (positive or negative). We can think of a test suite as a set of parameterized tests, where each such parameterized test is equipped with a set of parameter mappings $F = \{f_1, \ldots, f_n\}$. Applying each f_i to *pt* will yield a concrete test trace t_i.

4.2 Separation

Section 5 will explore the abstraction mechanism, but it is easy to see that an abstract representation could be more accurate when working on as large a number of examples as possible. An abstraction that only takes into account the values obtained by the parameter mappings attached to a certain parameterized test may result with a small number of concrete samples. This may yield an abstract

representation which is too coarse. Even worse, the abstract representation may provide no generalization.

To address this, we introduce another definition relying only on the statements in the test trace:

Definition 2 (Scenario). *A* scenario *S is a set of parameterized tests which execute the same sequence of statements, differing only by their parameters.* The code of a scenario *S, is the sequence of statements mutual to all parameterized tests in S, after discarding parameter information. We say that a parameterized test pt* belongs *to a scenario S if the code of S is obtained by discarding pt's parameter information.*

Continuing our example with $pt_1 =$ `assert(Precision.equals(x, x, y));`, if S is the scenario to which pt_1 belongs, then the code of S is the statement `assert(Precision.equals(?, ?, ?));` (without parameter information).

The unification of parameterized tests into scenarios is driven by the fact that despite the different parameter mappings, they are all running the same code (It is important to note that method overloading information is not discarded.)

Next, we formalize subsumption between parameterized tests of the same scenario. These definitions will allow us to increase the number of parameter mappings that can be attached to a single parameterized test.

To define subsumption, we wish to compare two parameterized tests from the same scenario and assess their generality. To do that, we need to compare the parameter uses in the parameterized test in sequence. We therefore rely on the following definition:

Definition 3 (Sequence of parameters). *Given a parameterized test pt, let params(pt) be the sequence of parameters across all statements in the parameterized test pt (with repetitions).*

This notion is needed so that we may compare two parameterized tests in the same scenario with a different number of parameters or with equality constraints in different places in the test trace. E.g., for $pt =$`foo(x,y);assert(bar(x,z));` we have $params(pt) = x \cdot y \cdot x \cdot z$.

Definition 4 (generality of parameterized tests, $\sqsubseteq$). *For two parameterized tests pt_1, pt_2 with $params(pt_k) = x_1^k \cdots x_n^k$ for $k \in \{1, 2\}$, both belonging to the same scenario S, we say that $pt_1 \sqsubseteq pt_2$ if $\forall i, j \in \{1 \ldots n\}$:*

1. *$type(x_i^1) \sqsubseteq type(x_i^2)$ (we use the standard notion of this relation, e.g. $int \sqsubseteq double$, $String \sqsubseteq Object$.)*
2. *$name(x_i^2) = name(x_j^2) \Rightarrow name(x_i^1) = name(x_j^1)$*

The definition above allows us to create a parameter mapping f_2 for a parameterized test pt_2 from a parameter mapping f_1 for parameterized test pt_1, such that $pt_1 \sqsubseteq pt_2$. We do this by defining the result of f_2 for every $x_i^2 \in params(pt_2)$ by $f_2(x_i^2) = f_1(x_i^1)$.

The implication of the correctness of behavior described by all parameterized tests in a scenario is that all parameter mappings in a scenario can and should

be abstracted together. However, creating a single abstraction for the entire scenario will create a unification problem.

Example 1. Let us consider three parameterized tests that have several parameter mappings each: $pt_1 =$ `int prev = x.size(); x.add(y); assert(x.size() == prev + 1);` with $type(x) = List\langle String\rangle$ and $type(y) = String$, pt_2 is identical to pt_1 except for having $type(x) = ArrayList\langle String\rangle$, and pt_3 is identical to pt_1 except for having $type(x) = Set\langle String\rangle$. Since `add` and `size` are methods on `List` and `Set`'s shared parent interface `Collection`, pt_1, pt_2, and pt_3 all belong to the same scenario.

Since $params(pt_1) = params(pt_2) = params(pt_3) = x \cdot x \cdot y \cdot x$, and since `ArrayList` is a subtype of `List`, but `Set` and `List` only share a common ancestor, we see that $pt_2 \sqsubseteq pt_1$, and pt_3 is incomparable with both.

We notice that even though pt_1 and pt_3 are incomparable, there exists a parameterize test pt_4, with the same test code and $type(x) = Collection\langle String\rangle$ and $type(y) = String$ for which $pt_1 \sqsubseteq pt_4$ and $pt_3 \sqsubseteq pt_4$.

If we aim to abstract pt_4, we can see that our unification problem is twofold. First, we now need to abstract (and later generate values for) collections in general, not just lists and sets, from concrete data that only includes lists and sets. We also see that there is a difference in behavior between sets and lists in this test code which needs to be captured by the abstraction: for a set, $res \mapsto true$ only if y is not already a member of the set, whereas for a list (ArrayList or otherwise), $res \mapsto true$ always. This problem is made even worse in cases where the shared ancestor is *Object*.

In order to avoid these problems we set a unification rule as follows:

Definition 5 (Abstraction candidates). *Let $T \subseteq S$ be the set of parameterized tests in a scenario S such that for every $pt \in T$, $\neg\exists pt' \in S.pt \sqsubseteq pt' \wedge pt \neq pt'$. We define the abstraction candidates for S to be the sets of parameter mappings $AC_S = \{\{f \in pt' \mid pt' \sqsubseteq pt\} \mid pt \in T\}$. When performing abstraction, each $s \in AC_S$ will be abstracted on its own.*

In other words, given the DAG defined by the $\sqsubseteq$ relation, we create an abstraction for every root pt, including with it the parameter mappings of every parameterized test reachable from pt. This means we only create abstractions for parameterized tests that exist “in the wild”, whilst reusing as many test traces as possible in order to abstract them.

5 Abstracting the test data

In the following section, we abstract each of the sets of examples in AC_S.

Once a parameterized test has its final set of concrete test traces, the *res* parameter can be used to divide them into positive and negative samples. For instance, the parameterized test for `Precision.equals(x,y,z)` with $type(x) =$

$type(y) = type(z) = double$ from Fig. 1 has the following data:

$$C^+ = \{(153.0, 153.0, .0625), (153.0, 153.0625, .0625), (152.9375, 153.0, .0625)\},$$
$$C^- = \{(153.0, 153.0625, .0624), (152.9374, 153.0, .0625)\}.$$

Safe Generalization We are interested in an abstraction in *some* language that would be a Safe Generalization [42], or an abstraction that provides separation from a set of counterexamples. Safe Generalization is defined as an operation that further generalizes a set of abstract elements A from an abstract domain [14] into another set of abstract elements, A', while avoiding a set of concrete counterexamples C_{cex}, and provides the following properties:

1. **Abstraction**: A' contains every concrete element that is abstracted by A (even though $A \sqsubseteq A'$ may not hold)
2. **Separation**: No $c \in C_{cex}$ is abstracted by A'
3. **Precision**: Generalization is a direct result of the elements in A.

as well as a strive for **maximality** that is not relevant for this use. We wish to generate two properties for the parameterized test that we are abstracting: one expecting the test to succeed, and one expecting it to fail. The code in these two properties is the same except for a negation of the assertion, but they require different data generators. In order to create these two generators, we need two abstractions, A^+ for the positive examples of the parameterized test, and A^- for the negative.

It is important to notice that, when a scenario has multiple abstraction candidate sets, they are still all representing the same behavior in the code under test, which means they are influenced by the counterexamples in the other sets as well. Specifically, while the positive examples were separated by the unification rule, and should not be abstracted together, they should still be separated from every negative point in the scenario, as they all represent some negative case for the same code. This applies symmetrically to the negative points.

We therefore define for an abstraction candidate $a \in AC_S$ the following example sets:

$$C^+ = \{f \in a | f(res) = +\} \qquad C^- = \{f \in a | f(res) = -\}$$
$$C^+_{cex} = \bigcup_{b \in AC_s} \{f \in b | f(res) = -\} \qquad C^-_{cex} = \bigcup_{b \in AC_s} \{f \in b | f(res) = +\}$$

and attempt to attain the separating abstraction for A^+ from (C^+, C^+_{cex}) and for A^- from (C^-, C^-_{cex}).

It is clear that not every abstraction language will be able to accommodate this requirement. In addition, when there are few samples, many different elements from each language may fit, and we are not necessarily interested in the most precise one, which means we will need to relax the precision requirement of Safe Generalization.

In some domains such as Intervals, we are able to easily compute Safe Generalization using algorithms such as Hydra [36], but we may still wish to perform

a controlled loss of precision on the result. In other domains, computing Safe Generalization will be doubly exponential. Instead, we utilize a Safe Generalization relation, denoted $SG(C, C_{cex})$, which includes the set of abstractions that are safe generalizations for $(\{\beta(c) \mid c \in C\}, C_{cex})$, where β is the abstraction function for a single concrete element. SG relaxes the precision requirement, allowing abstractions to be included in $SG(C, C_{cex})$ even for very small example sets C, C_{cex}.

In theory we would construct SG over every available abstraction. In practice, we use SG to test a set of given abstractions.

Abstraction templates In order to select an abstraction language and an element of that language, JARVIS contains a library of abstraction templates, such as $|x - y| \leq z$, $x \in [a, b]$, etc. As previously shown by the FlashFill project [48], the case of learning from few examples (in FlashFill, often only one example) requires the notion of ranking the possible programs, or in our case, possible abstractions, so that correct programs will be ranked higher than incorrect ones, and likely programs higher than unlikely ones. While in [48] this ranking is learned from examples, in our implementation the templates have a predefined ranking that is applied for all instantiated abstractions that hold for all samples.

Every template t of the template library is instantiated, and in the case of templates such as $x \in [a, b]$ or $|a * x - y| \leq b$, the parameters are selected based on the existing samples. Templates are instantiated in pairs, one as an abstraction for the positive examples and one for the negative. The result is $\mathcal{A} = \{(A^+, A^-) | (A^+, A^-) \in SG(C^+, C^+_{cex}), (A^-, A^+) \in SG(C^-, C^-_{cex})\}$. We then select from $\mathcal{A}$ the highest ranking (A^+, A^-), and create code sampling them as the generators for the properties.

This means the template library can be extended to include more abstractions, and the ranking can be modified to better suit a specific project or domain.

5.1 Handling Impreciseness

The abstractions we use are conservative, and overapproximate the concrete data that they abstract. On the one hand, this guarantees that cases that are present in the original unit test will be included in the generated PBTs. However, in some cases, even the best abstraction available in the template library will be too conservative, and also represent data points that will fail the PBT. This can happen for one of two reasons:

- The abstraction itself is not precise enough (e.g. a single interval, when the data requires a disjunctive abstraction, or a set of intervals).
- The number of examples is too low to precisely generalize from (e.g. generalizing from two examples, there is not enough data to reduce the set of abstraction templates).

Both cases require manual intervention: in the first case, the user can provide a finer abstraction, in the second case she can provide more examples, and in either case she can manually edit the resulting tests.

6 Sampling from the Abstraction

We now wish to sample the abstraction that was created in the previous section. When creating the abstraction-based value generators that will sample the abstraction, we take our cue from the original test traces and their parameterized tests. We consider the original tests written by the programmers to be a weighted sample from the region of the domain that is described by the "true" precondition of the tested behavior. That is, the user has already selected points that they deem important. We therefore wish to preserve them.

We have created an abstraction of each region – an underapproximation of the positive and negative regions for each of the maxima of the $\sqsubseteq$ relation. We now wish to generate property-based tests, or in essence, to generate code that will sample from the concretization of our abstraction. The sampling component of the code in Fig. 2 is shown in lines $1-6$ and $11-17$. It is composed of the representation of the space and types of the variables to be sampled into.

In this section we describe the creation of such sampling for the abstractions we performed.

Sampling based on user-encoded testing nuances We notice that we may wish to sample each abstracted region more than once. Since the constraints of parameterized tests lower in the hierarchy represent constrained values sampled by the user, we wish to cover them in our generated sampling. Let us examine the parameterized test $pt_1 =$ `assert(Precision.equals(x, x, y));` with $type(x) = type(y) = double$ from Section 2. It is sampled out of the region abstracted for the entire scenario S containing pt_1 as well as other tests for `Precision.equals`. Abstracting the topmost parameterized test by the $\sqsubseteq$ relation yields an abstraction in $\mathbb{R}^3$. When sampling $(x, y, z) \in A \subseteq \mathbb{R}^3$ the odds of satisfying the constraint in pt_1, i.e., $x = y$, are infinitesimal. If we wish to preserve the constraint entered by the user, we must sample the special case in which $x = y$ on its own.

Sampling the constraints In order to sample each set of constraints on its own, we create a sampling component as follows: for every $pt \in S$, we create a sampling component over each abstraction for an abstraction candidate s for which pt has contributed its parameter mappings. For each region sampled, the constraints of pt are added to the restrictions on the domain. For example, when sampling the region $|x - y| \leq z$ for pt_1 seen in Section 2, the new sampling constraints are $|x - y| \leq z \wedge x = y$, or $0 \leq z$, sampling (x, z) out of this region s.t. $type(x) = type(z) = double$.

Sampling guarantee Finally, we formulate our guarantee for points that will be sampled:

Claim. Let T be a set of test traces from the same scenario S, $|T| \geq 2$. For each $t \in T$, if $\exists t' \neq t$ s.t. $PT(t) \sqsubseteq PT(t')$, then

1. t will be used in an abstraction, and
2. $PT(t)$ will be used to create an abstraction-based value generator for a PBT.

Library	Scenarios: avg size	Scenarios: repeating scenarios	Repeating scenarios: no. of traces avg	no. of traces max	no. PTs per scenario avg	no. PTs per scenario max	have multiple PTs	Hierarchy: height avg	height max	no. of roots avg	no. of roots max
Commons-CLI	4.2	38.3%	3.5	14	1.067	2	6.7%	1.033	2	1.50	2
Commons-Codec	2.2	38.2%	3.4	8	1.088	2	8.8%	1.088	2	1.00	1
Commons-Collections	2.1	13.3%	2.8	5	1.133	4	6.7%	1.033	2	2.50	3
Commons-Configuration	3.5	14.6%	3.7	15	1.015	2	1.6%	1.012	2	1.25	2
Commons-CSV	3.2	27.8%	2.0	2	1	1	0.0%	1.000	1	1.00	1
Commons-Email	2.3	60.0%	2.7	5	1.1	2	10.0%	1.100	2	1.00	1
Commons-IO	2.9	23.8%	3.6	14	1.069	4	4.7%	1.042	2	1.11	2
Commons-JEXL	4.6	25.9%	2.4	4	1.037	2	3.7%	1.000	1	2.00	2
Commons-Lang	2.1	36.7%	5.5	37	1.273	5	19.1%	1.212	3	1.04	3
Commons-Math	4.3	19.7%	4.1	45	1.182	9	10.2%	1.075	4	1.79	6
Commons-Pool	3.9	33.3%	2.0	2	1.222	2	22.2%	1.222	2	1.00	1
Commons-Text	2.5	36.7%	6.2	15	1.133	2	13.3%	1.133	2	1.00	1

Table 1. Scenario makeup of the JUnit test suites of Apache-Commons projects. Repeating scenarios are those with the number of concrete test traces greater than 1.

This allows for a maximal reuse of examples for abstraction, and on the other hand, the sampling of all special cases that are abstracted.

7 Experimental evaluation

We implemented JARVIS to operate on JUnit test suites written in Java and to synthesize ScalaCheck PBTs. Scala has a seamless interoperability with Java [38], which means properties for ScalaCheck, which are written in Scala, can mimic completely the functionality of the original test traces. JARVIS uses the Polyglot compiler [37] and the ScalaGen [5] project to translate test traces from Java to Scala. Template instantiation is aided by the Z3 SMT solver [15].

We ran JARVIS on the test suites of several open source libraries. We tested whether the hierarchy and unification rule of abstraction candidates are relevant to real-world test suites.

7.1 Examining Apache test suites

Tab. 1 shows the result of running JARVIS on the test suites of 12 Apache Commons projects. This summary of JARVIS's ability to unify shows us several things in regard to the problems it addresses:

Identifying tests In all projects, the average length of extracted test traces is over two statements. This shows JARVIS identifies and extracts elaborate tests.

Repetition of tests The data shows that there is, in fact, enough repetition of tests to justify test generalization. In the project with the least amount of repetition, Commons-Collections, only 13% of the scenarios contain more than one concrete test trace, but in half the projects this number is over 30%.

Existence of constraints When examining the scenarios that contain more than one test trace, we see that this is, on average, 9% of scenarios, with as many as 9 separate parameterized tests, or sets of constraints, in the same scenario.

These are all user-encoded sampling constraints that would see their probability plummet without a specific sampler generated for them.

Importance of the unification rule In scenarios that have multiple parameterized tests, we see that the number of roots in the hierarchy DAG (i.e., incomparable maxima of the relation) is, on average, 1.35. This means that it is not infrequent to have scenarios where the least upper bound of two or more parameterized tests does not occur in the test traces. Since this is a frequent occurrence in the real world, we deem it frequent enough to address the issues that arise from over-unification within the scenario (as described in Section 4.2).

7.2 Increased coverage

The extended version of our paper contains coverage experiments comparing JARVIS-generated PBTs to the original test suite from which it was generated. Scenarios from Apache Commons-Math and Commons-Lang were included.

Instruction coverage, or the number of lines of code in the library under test that were exercised by the test code, was preserved or marginally improved for all benchmarks. *Value coverage*, however, was sometimes increased by up to two orders of magnitude. This increase is especially important because of the ability to find bugs that are not "boundary values" in terms of the structure of the program, like the one described in Section 8.

Additionally, while running these experiments we managed to identify a bug in the Common-Lang test suite, and our submitted repair[1] was accepted.

8 Discovering Bugs: A Case Study

In this section we review a historical bug in Apache Commons-Math that we discovered by running JARVIS on the unit test suite for the version before the bug was fixed.

The extended version of this paper includes a second bug found by JARVIS: a critical severity bug in the `ContinuedFraction` class, discovered via the PBT generated from the unit tests for the `FDistribution` class[2]. Unlike the full case study presented here, this bug required manual intervention as described in Section 5.1.

8.1 MATH-1256: Interval bounds

In Apache Commons-Math versions prior to 3.6, the test suite for the `Interval` class included the code in Fig. 3. A bug in the interval class which is not tested in these unit tests was opened as "MATH-1256: Interval class upper and lower check"[3]. An Interval object could be created with a lower bound greater than its

[1] http://github.com/apache/commons-lang/pull/230
[2] http://issues.apache.org/jira/browse/MATH-785
[3] http://issues.apache.org/jira/browse/MATH-1256

```
1  @Test
2  public void testInterval() {
3      Interval interval = new Interval(2.3, 5.7);
4      Assert.assertEquals(3.4, interval.getSize(),
5          1.0e-10);
6      Assert.assertEquals(4.0, interval.getBarycenter(),
7          1.0e-10);
8      Assert.assertEquals(Region.Location.BOUNDARY,
9          interval.checkPoint(2.3, 1.0e-10));
10     //Other asserts on properties of interval
11 }
12 //...
13 @Test
14 public void testSinglePoint() {
15     Interval interval = new Interval(1.0, 1.0);
16     Assert.assertEquals(0.0, interval.getSize(),
17         Precision.SAFE_MIN);
18     Assert.assertEquals(1.0, interval.getBarycenter(),
19         Precision.EPSILON);
20 }
```

Fig. 3. Unit test code for the Interval class from the Apache Commons-Math project

```
1  val gen_double_1_pos  = for(
2      x <- Gen.posNum[Double];
3      y <- Arbitrary.arbitrary[Double];
4      z <- Gen.oneOf(y-x,y+x)
5          .suchThat(t => Math.abs(t-y) == x)
6      ) yield (x,y,z)
7
8  forAll (gen_double_1_pos) {_ match {
9  case (double_1,double_3,double_2) =>
10   val interval = new Interval(double_1, double_2)
11   double_3 ~= interval.getSize
12 }}
```

Fig. 4. The ScalaCheck generator and property generated by JARVIS from the unit tests in Fig. 3.

upper bound, which would result in an invalid interval with a negative size. The bug report shows test code initializing `Interval interval0 = new Interval(0.0, (-1.0));` and showing that it would result in `interval0.getSize()` being -1.0.

This bug hinges on the two parameters accepted by the `Interval` constructor. Since it only requires $y > x$, it exists in nearly 50% of the parameter space. However, the conventional unit tests in `IntervalTest` only cover 2 values.

Running JARVIS on `IntervalTest.java` from release 3.5 yields 9 different scenarios. The scenario testing `getSize` contains two parameterized tests, one for the parameters $double_1$, $double_2$ and $double_3$, from the code in `testInterval` and one for $double_1$ and $double_2$ from the code in `testSinglePoint`. We denote them pt_3 and pt_4, respectively. Since $pt_4 \sqsubseteq pt_3$, the concrete test from `testSinglePoint` is added to the parameterized test for `testInterval`, resulting in $C^+ = \{(2.3, 5.7, 3.4), (1.0, 1.0, 0.0)\}$.

From the abstraction template library for 3D abstractions, the abstraction selected for these points by the criteria outlined in Section 5 is $|y - z| = x$.

JARVIS outputs the code in Fig. 4 to generate values matching the abstraction. Running the test with ScalaCheck fails in cases where the upper bound of the interval is negative while the lower bound is generated as always positive. Since the bug exists in nearly 50% of the space, it occurs almost immediately when running the PBT. These cases expose MATH-1256 without the additional unit tests that were later added after it was reported and fixed.

9 Related Work

Learning from examples Learning from examples or "Programming by Example" is a field of synthesis with many different applications, such as Inductive Programming [18], string processing [28, 27, 48] and data extraction [31]. In particular, the FlashFill and FlashExtract projects [48, 31] present an interactive algorithm for synthesis by examples used to generate code for string manipulation, showing that it is possible to synthesize a program from few examples, despite having several compatible solutions.

Genrating Tests and Oracles An experiment described in [46] reveals that state of the art automatic test generation tools are far from satisfactory. However, an experiment described in [44] shows that manual unit tests, written by developers aided by an automatic test generation tool, create better code coverage. [40] use code instrumentation of the system under test to guide test generation by path discovery. [39] suggest a technique that improves random test generation by avoiding sequences calls after an object has reached an illegal state. [57] extend this technique further to increase coverage and diversity for automatically generated tests. However, their method only generates the tests and they do not suggest how to generate oracles. [22] describe a technique for automatically generating unit tests together with appropriate pre and post-conditions, based on mutations of the tested class and test inputs. However, the basis for the postcondition is the observable state after the test execution, which means that bugs in the program will result in incorrect postconditions. [24] suggests a mutation-based technique to select variables for which an oracle would detect a high number of faults in the program. However, a tester is still required to write the oracles. [58] generate unit tests using symbolic execution and incremental refinement. [54] generate both tests and oracles from use case specifications, using natural language processing techniques.

Parameterized Unit Tests PUTs are defined in [52] and as "theory-based testing" in [45] and developed further in [51, 53, 55]. [50] is an empirical study in unit test parametrization that strongly advocates parameterized unit testing. Generalizing unit tests to parameterized unit tests was shown as useful in detecting new bugs and required feasible human effort, though one that required expertise with additional tools. However, their proposed methodology contains manual steps for parametrization and generalization, and they do not address the problem of extracting and grouping the tests. [23] extends [22] to parameterized unit tests, but exactly as in [22], the postcondition is derived from the

observable state after the test execution. In contrast, JARVIS creates test oracles from the oracles of the original unit tests, and treats their assertions as part of the generalization, making no assumptions based on the execution.

Property Based Testing and Fuzzing [21] creates PBTs for web services, but does so from both a syntactic and manually-written semantic description of the service. Later work [32] is intended to track API changes in web services and update existing PBTs. [25] recognizes the important connection between conventional unit tests and PBTs, and describe a tool that checks whether a given unit test is covered by a given PBT. [56] is a tool for automatic PBT generation, based on feedback directed random test generation [39]. However, similar to previous feedback directed random test generation techniques, the oracles are specified by the developer. Fuzz testing or "fuzzing", another testing technique, is very similar to property-based testing: it draws inputs or enumerates them, but usually does not use oracles, only looks for crashes, and does not test components but rather the whole program. Works such as [9] draw their inputs from grammars of valid or invalid inputs. Others add to this a white-box approach [26, 35], attempting to draw inputs that increase coverage. [12] suggest combining fuzzing with ranking, based on the diversity of the test cases. [6] aim to draw inputs for fuzzing that will direct the fuzzing to a part of the program.

10 Conclusion

We presented JARVIS, a tool to extract repetitive tests from unit test suites and synthesize from them property-based tests. We have shown the foundations for its operation: sorting the existing unit tests into sets of compatible tests; a better abstraction, achieved using a hierarchy of generality between parameterized tests which allows abstractions to generalize more tests; generalizing the examples to a data generator, taking into consideration the positive and negative examples (tests expected to succeed and fail, resp.) using the notion of Safe Generalization; and preserving the subtleties of human-written unit tests by sampling each abstracted region according to constraints found in the test data.

We applied JARVIS to the JUnit test suites of 12 Apache Commons APIs, and have shown there is ample repetition in the data of real-world test suites, which can be used to generate PBTs. We have also shown that the repetition often includes subtleties, in the same testing scenario. Additionally, we have shown that JARVIS-generated PBTs maintain the instruction coverage of the original unit tests, and increase parameter value coverage by as much as two orders of magnitude. PBTs generated by JARVIS have found a known bug in Apache Commons-Math, and with the help of JARVIS we identified a bug in the Commons-Lang test suite.

Acknowledgements

The research leading to these results has received funding from the European Unions Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME.

References

1. Jsverify: Write powerful and concise tests.
2. Junit. http://junit.org/.
3. Nunit. http://www.nunit.org/.
4. Scalacheck: Property-based testing for scala. http://www.scalacheck.org/.
5. Scalagen: Java to scala conversion.
6. M. A. Alipour, A. Groce, R. Gopinath, and A. Christi. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 70–81. ACM, 2016.
7. D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina. Automatic test generation for coverage analysis using cbmc. In *International Conference on Computer Aided Systems Theory*, pages 287–294. Springer, 2009.
8. T. Ball. A theory of predicate-complete test coverage and generation. In *International Symposium on Formal Methods for Components and Objects*, pages 1–22. Springer, 2004.
9. R. Brummayer and A. Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5. ACM, 2009.
10. J. R. Calamé, N. Ioustinova, and J. van de Pol. Automatic model-based generation of parameterized test cases using data abstraction. *Electronic Notes in Theoretical Computer Science*, 191:25–48, 2007.
11. M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in focaltest. In *ICSOFT*, 2010.
12. Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 197–208, New York, NY, USA, 2013. ACM.
13. K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.
14. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
15. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
16. J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-A. Fredlund, V. Gulias, J. Hughes, and S. Thompson. Property-based testing-the protest project. In *International Symposium on Formal Methods for Components and Objects*, pages 250–271. Springer, 2009.
17. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.
18. C. Ferri-Ramírez, J. Hernández-Orallo, and M. Ramírez-Quintana. Incremental learning of functional logic programs. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *LNCS*, pages 233–247. Springer-Verlag, 2001.
19. G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.
20. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal*

Methods for Increasing Software Productivity, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.
21. M. A. Francisco, M. López, H. Ferreiro, and L. M. Castro. Turning web services descriptions into quickcheck models for automatic testing. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, pages 79–86. ACM, 2013.
22. G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.
23. G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374. ACM, 2011.
24. G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl. Automated oracle data selection support. *Software Engineering, IEEE Transactions on*, 41(11):1119–1137, 2015.
25. A. Gerdes, J. Hughes, N. Smallbone, and M. Wang. Linking unit tests and properties. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*, pages 19–26. ACM, 2015.
26. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.
27. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.
28. S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012.
29. D. Hoffman, P. Strooper, and L. White. Boundary values and automated component testing. *Software Testing, Verification and Reliability*, 9(1):3–26, 1999.
30. J. Hughes. Quickcheck testing for fun and profit. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–32. Springer, 2007.
31. V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In M. F. P. O'Boyle and K. Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.
32. H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco. Automating property-based testing of evolving web services. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 169–180. ACM, 2014.
33. N. P. Lopes and J. Monteiro. Weakest precondition synthesis for compiler optimizations. In *Verification, Model Checking, and Abstract Interpretation*, pages 203–221. Springer, 2014.
34. A. Löscher and K. Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 46–56. ACM, 2017.
35. R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 134–143. ACM, 2007.
36. K. S. Murray. Multiple convergence: An approach to disjunctive concept acquisition. In *IJCAI*, pages 297–300. Citeseer, 1987.
37. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction*, pages 138–152. Springer, 2003.
38. M. Odersky, L. Spoon, and B. Venners. Scala. *URL: http://blog. typesafe. com/why-scala (last accessed: 2012-08-28)*, 2011.

39. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.
40. R. Pandita, T. Xie, N. Tillmann, and J. De Halleux. Guided test generation for coverage criteria. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.
41. M. Papadakis and K. Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 39–50. ACM, 2011.
42. H. Peleg, S. Shoham, and E. Yahav. D3: Data-driven disjunctive abstraction. In *Verification, Model Checking, and Abstract Interpretation*, pages 185–205. Springer, 2016.
43. L. Pike. Smartcheck: automatic and efficient counterexample reduction and generalization. In *ACM SIGPLAN Notices*, volume 49, pages 53–64. ACM, 2014.
44. J. M. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 338–349, New York, NY, USA, 2015. ACM.
45. D. Saff, M. Boshernitsan, and M. D. Ernst. Theories in practice: Easy-to-write specifications that catch bugs. 2008.
46. S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 201–211. IEEE, 2015.
47. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification*, pages 88–105. Springer, 2014.
48. R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification*, pages 398–414. Springer, 2015.
49. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *ACM Sigplan Notices*, volume 44, pages 223–234. ACM, 2009.
50. S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 294–309, Berlin, Heidelberg, 2011. Springer-Verlag.
51. N. Tillmann and P. de Halleux. Parameterized unit testing with pex (tutorial). In *Proc. of Tests and Proofs (TAP'08)*, volume 4966, page 171181, Prato, Italy, April 2008. Springer Verlag.
52. N. Tillmann and W. Schulte. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 253–262. ACM, 2005.
53. N. Tillmann and W. Schulte. Parameterized unit tests with unit meister. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 241–244. ACM, 2005.
54. C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 385–396, New York, NY, USA, 2015. ACM.
55. T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mutation analysis of parameterized unit tests. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 177–181. IEEE, 2009.

56. K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. Arbitcheck: A highly automated property-based testing tool for java. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '14, pages 405–412, Washington, DC, USA, 2014. IEEE Computer Society.
57. K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. Feedback-controlled random test generation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 316–326, 2015.
58. H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara. Fsx: Fine-grained incremental unit test generation for c/c++ programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 106–117. ACM, 2016.

A Logical System for Modular Information Flow Verification

Adi Prabawa, Mahmudul Faisal Al Ameen, Benedict Lee, and Wei-Ngan Chin

National University of Singapore
adi.prabawa@u.nus.edu mahmudulfaisal@gmail.com
dcsljhb@nus.edu.sg chinwn@comp.nus.edu.sg

Abstract. Information Flow Control (IFC) is important to ensure secure programs where secret data does not influence any public data. The pervasive standard that IFC aims to is non-interference. Current IFC systems are separated into dynamic IFC, static IFC, and hybrids between static and dynamic. With dynamic IFC suffering from high overhead and limited ability to prevent implicit flows due to the paths not taken, we propose a novel modular static IFC system. To the best of our knowledge, this is the first modular static IFC system. Unlike type-based static IFC systems, ours is logic-based. The limitation of type-based IFC systems is in the inviolability of static security label declarations for fields. As such, they suffer from transient leaks on fields. Our proposed system uses a Hoare-like logic. It verifies each function independently with the help of separation logic. Furthermore, we provide the proof of correctness for our novel IFC system with respect to termination- and timing-insensitive non-interference.

1 Introduction

Information Flow Control (IFC) tracks the propagation of information through a program. IFC is used to ensure that secret data does not influence public data. In other words, an attacker cannot infer secret data by observing any public data. This independence between secret and public data is the semantic notion proposed by Goguen and Meseguer as *non-interference* [13].

In many IFC systems, information release policies are based on a lattice of security labels [12, 23, 29]. In the context of non-interference, it means that information cannot flow from a higher security label to lower security label in the lattice. Given a piece of information and its corresponding security label, it must have been influenced by information from lower security labels.

The way information flows in a program is through assignment operations. This is called explicit flow. Sabelfeld and Myers [24] also classifies six covert channels based on the definition by Lampson [16] where information may flow from. Out of the six channels mentioned by Sabelfeld and Myers, the commonly tackled channels in information flow analysis are implicit flows, termination channels, and timing channels.

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 430–451, 2018.
https://doi.org/10.1007/978-3-319-73721-8_20

In implicit flows, information flows through control structures. Termination channels propagate information through the termination or non-termination of a computation while timing channels propagate information through how long a program is run.

Our IFC system handles only explicit and implicit flows while excluding termination and timing channels. This is called termination- and timing-insensitive non-interference. In other words, our system ensures that if the program terminates, it will have no information leakage.

Additionally, our system assumes a strong capability of attacker. In particular, we assume that the attackers may change all public input arguments and may see all resulting public values. Furthermore, to the best of our knowledge, our system is the first modular verifier for information flow analysis with recursion. It is modular as each function is checked independently of the context in which it is called.

There are various types of IFC. Static systems statically verify non-interference through type systems [18, 24] or program logic [4, 5, 11]. Dynamic systems uses run-time security labels attached to values and propagate them during program execution [8, 10]. Hybrid systems that combine static and dynamic analysis have also been studied [28]. Interestingly, IFC can be transposed into a safety problem as shown by Barthe *et al.* [3] and Terauchi and Aiken [27].

2 Motivation

Although IFC systems are based on a lattice of security labels, two levels, `Hi` and `Lo`, where `Lo` $\sqsubseteq$ `Hi` are sufficient for illustration. Given a lattice of security labels, we can enumerate all pairs of security labels and designate them as either `Hi` or `Lo` accordingly. Hence, we will only consider the two levels `Hi` and `Lo` in this paper.

Recent advances in information flow analysis have focused on dynamic analysis due to its potential use in Javascript [8, 10, 25]. However, dynamic IFC systems suffer from a large run-time overhead and they cannot prevent implicit flows due to paths not taken since propagation of security labels is done at run-time. While there are advances on the latter problem such as no sensitive upgrade and permissive upgrade [6, 7, 32], they rely on static analysis or crowdsourcing [15].

The problem with dynamic propagation of security labels can be seen in the example below. Consider an object o with four fields w, x, y, and z such that only x and w are declared as `Hi`. Suppose $o.x$ is either True or False, the value of $o.z$ will equal $o.x$ at the end. Information from $o.x$ flows to $o.z$ through $o.y$ implicitly via the path not taken (*i.e.,* via the else branch).

However, the security label of $o.x$ is not propagated to $o.z$. In the case of $o.x =$ True, the propagation from $o.x$ to $o.y$ does not occur since the statement $o.y :=$ True is not executed. In the case of $o.x =$ False, although propagation from $o.x$ to $o.y$ occurs, the propagation from $o.y$ to $o.z$ does not occur since the statement $o.z :=$ True is not executed. As such, in both scenarios, the security label of $o.x$ is not propagated to $o.z$.

```
o.y := False; o.z := False; o.w := True;
if o.x == False then o.y := True; else o.w := False;
if o.y == False then o.z := True; else o.w := False;
```

In static analysis, the most prevalent analysis method is using a type system [14, 18, 23, 24, 28, 29]. Type-based IFC declares security labels of fields statically and checks for violations to non-interference. Violations occur when information from higher security labels flow into the fields with lower security labels. In type-based IFC, a field with static security label `Lo` can never be assigned a value with security label `Hi` even when it is going to be immediately reassigned to another value with security label `Lo`.

As such, type-based IFC systems except for the system by Hunt and Sands [14] are flow-insensitive on object fields. Consider `y.f := x; y.f := 0;`, if $y.f$ is statically declared with security label `Lo` and x with `Hi`, the first assignment `y.f := x` violates the static security label declaration. This is despite having the value of $y.f$ rewritten immediately in $y.f := 0$. Thus, type-based IFC systems suffer from a high rate of false positives.

In logic-based IFC systems, Amtoft and Banarjee [4] shows how analysis on the independence of program variables, as opposed to dependence, can be used to analyze information flow on simple programs. In their system, they use a Hoare-like logical system with programming language syntax based on the While language without functions. This system is then extended to analyze Java-like programs [5]. However, it resulted non-modular interprocedural analysis.

Costanzo and Shao gave a logic-based IFC system with a semantic-based declassification policy and dynamic context lifting [11]. Similar to Amtoft and Banarjee, their language is based on the While language without functions. Hence, program verification in their system is done on the entire program even though it supports separation logic. The frame rule introduced in separation logic [1, 9, 21, 26] allows for modular verification. In short, separation logic is a logic with separation conjunction. Separation conjunction is used to reason about shared mutable resources such that verification can be done modularly.

2.1 Contribution

In this paper, we introduce a novel modular logic-based IFC system. Our goal is to ensure termination- and timing-insensitive non-interference in a fully modular way. The basic idea of our system is to abstract the lattice of security labels into security bounds. The security bound of a value written as $x <: \varsigma_x$ intuitively means that the security level of x is at most ς_x. A flow from x to y is then represented as a transfer of the security bound from one value to another.

To achieve modularity, our programming language is extended with function calls and function definitions. Every function definition is annotated with a precondition and postcondition that includes separation logic and a security formula consisting of security bounds. The details of the syntax and semantics of the language is presented in Section 3.

Verification of functions proceeds by assuming the precondition to obtain the post-state that is used to entail the postcondition. Therefore, the verification is

modular as each function can be verified independently using only the stated precondition and postcondition. In short, the essence of our system is to find the sound approximation of the security bounds of all the values in a function. This information is then used to entail the postcondition. To the best of our knowledge, this is the first modular information flow analysis system.

Being a static analysis system, our system does not suffer from the high overhead incurred in dynamic systems. Furthermore, as static systems consider all the possible paths including the paths not taken, our system can handle implicit flow easily. In comparison with a type-based system, our system allows flow-sensitive analysis even on object fields. As such, temporary assignments to higher security labels simply transfer the bounds and do not trigger an error.

In summary, the main contribution of our paper is as follows:

1. A novel logic-based IFC system that allows for modular verification via pre and postconditions containing security formulas. Our system is modular as verification of functions are independent of the context in which the function is called.
2. A proof of correctness for the novel IFC system introduced in this paper. The proof of correctness uses observation equivalence or bisimulation equivalence as defined by Milner [17] and Park [20].

Additionally, as a minor contribution, our system handles aliasing directly through the use of separation logic. As an added advantage, our system overcomes the limitations of dynamic IFC, namely high run-time overhead, dynamic security label propagation. Furthermore, unlike type-based IFC, our system is flow-sensitive even on object fields.

3 Language

In this section, we discuss our programming and assertion languages. Our programming language is an expression-oriented pointer programming language. We assume that programs written are well-typed for simplicity. We denote variables, reference variables, and constants by v, r and c respectively. We use r exclusively for reference variables. Class names (or structures), fields, and function names are denoted by C, i, and f respectively. We also denote the binary operators $=$, $\leq$, $+$, and $-$ by $\diamond$.

In our language `while` statements are represented by recursive functions with reference parameters. A variable v can be either local, l or reference, r. Global variables are also represented by references.

Definition 1. *An expression in our language is defined as below.*

$$e ::= c \mid v \mid e_1 \diamond e_2 \mid \text{new } C(\overrightarrow{v}) \mid v.i \mid f(\overrightarrow{r}, \overrightarrow{v}) \mid v := e \mid \text{if } v \text{ then } e_1 \text{ else } e_2 \\ \mid v_1.i := v_2 \mid e_1; e_2$$

The expression new $C(\overrightarrow{v})$ is like *malloc* of the C language except that it requires initialization of all its fields by $\overrightarrow{v}$ and it returns an instance of the class (or structure) C. We consider classes to consist of only fields and not functions. In other words, functions are independent of the classes. We represent $v.i$ as the field i of the instance v of some class C where i is a declared field of C.

Definition 2. *Our assertion language is defined as below.*

$$\Delta ::= \Theta \wedge \Phi \qquad \Theta ::= \bigvee(\exists \overrightarrow{v}.\Pi \wedge \Sigma) \qquad \Sigma ::= emp \mid v \mapsto \langle \overrightarrow{i : y} \rangle \mid \Sigma_1 * \Sigma_2$$
$$\Pi ::= a \mid \neg \Pi \mid \Pi_1 \wedge \Pi_2 \mid \Pi_1 \vee \Pi_2 \mid \Pi_1 \to \Pi_2 \mid \exists y.\Pi \mid \forall y.\Pi$$
$$a ::= d_1 = d_2 \mid d_1 \leq d_2 \qquad d ::= c \mid v \mid c \times d \mid d_1 + d_2 \mid -d$$
$$\Phi ::= v <: \varsigma \mid v.i <: \varsigma \mid \theta <: \varsigma \mid \Phi_1 \wedge \Phi_2 \qquad \varsigma ::= \texttt{Hi} \mid \texttt{Lo}$$

Our program specification is given as a separation logic formula extended with a security formula. We use the heap formula $v \mapsto \langle \overrightarrow{i : y} \rangle$ to indicate an object v with fields $\overrightarrow{i}$ such that each field i has a value y. A security formula describes a set of variables with their respective security upper bounds. Security bounds are defined by ς that is a lattice of security labels with s denoting an element of this lattice. We use ς_v to denote the security bound of variable v. Every variable is bounded by this security label as shown in $v <: \varsigma_v$. For simplicity, we consider only Hi and Lo where $\texttt{Lo} \sqsubseteq \texttt{Hi}$ as the ς in this paper. In theory, our system works for any lattice of security labels. In particular, we can consider every pair (s_1, s_2) of security label in a lattice such that if $s_1 \sqsubset s_2$, then for all s_h such that $s_2 \sqsubseteq s_h$, we set it to be Hi. Similarly, for all s_l such that $s_l \sqsubseteq s_1$, we set it to be Lo.

For the pure formula, we consider Presburger arithmetic for its decidability. θ is another special variable to capture the result or output value and security upper bound of an expression. The symbol θ indicates the security bound of the result of evaluating an expression.

Informally, $v <: \texttt{Lo}$ means v must be public data and $v <: \texttt{Hi}$ means v can be either public or secret data. For example, if a field f of an object x has a value y and a security bound Lo, we can write $x \mapsto \langle f : y \rangle * x.f <: \texttt{Lo}$. This allows our system to be practically applied to some real-world applications.

Although we have separating conjunction, we do not have separating implication. The main reason for this design choice is that our logical system consists of only the forward rules. So, separating implication will not be necessary. A program P consists of class declarations $\overrightarrow{C}$ followed by function definitions $\overrightarrow{F}$. A program is defined below.

$$P ::= \overrightarrow{A}\,\overrightarrow{F} \qquad A ::= \text{data } C\{\overrightarrow{i}\} \qquad F ::= f(\overrightarrow{r}, \overrightarrow{v})(\Delta)(\Delta')\{e\}$$

We say that a function declaration $f(\overrightarrow{r}, \overrightarrow{v})(A \wedge \Phi)(A' \wedge \Phi')\{e\}$ is well-defined if $\overrightarrow{r} \cup \{\theta\} = \mathrm{FV}(\Phi')$ where $\mathrm{FV}(\Phi)$ denotes the free variable of formula Φ. In other words, a well-defined function declaration has to capture all effects (*i.e.,* return value and reference variables) of the function. Note that when it appears in the

postcondition Δ', the symbol θ denotes the result of the function. However, in the source code, we use `res` instead of the symbol θ.

It is a typical requirement that the flow formula of Δ contains at least the parameters that have security upper bound `Lo`. Otherwise, in the absence of a parameter in the flow formula, `Hi` is considered as the default security upper bound. We consider call-by-reference for reference parameters and call-by-value for non-reference parameters in recursive functions as defined by Winskel [31]. The use of call-by-reference is needed to simulate iteration statements such as the `while` statement. Note that we do not have explicit variable declarations and we will assume that all the variables in a function body are declared implicitly at the beginning of the body.

We define our environment Γ as the set of asserted programs that consists at least all $\{\Delta\}f(\overrightarrow{v})\{\Delta'\}$ for all function declarations $f(\overrightarrow{v})(\Delta)(\Delta')\{e\}$ as in [1, 2]. In our system, judgements are in the form $\Gamma \vdash \{\Delta\}e\{\Delta'\}$ where Γ is the set of asserted programs.

Note that our language does not feature the annotation of a variable or a field with a security upper bound. By default, context always starts from `Lo` and so we need a starting point where we can write a program with variables upper bounded by `Hi`. This can be done by using the special function `secret (p) (p <: Lo) (res <: Hi) { p }`.

3.1 Semantics

The semantics of our programming language and assertion language are discussed here. Our semantics are influenced by Costanzo's paper [11] and is based on [31]. The set $\mathbb{N}$ is defined as the set of natural numbers. The set Vars is defined as the set of variables. The set Locs is defined as the set $\{n \in \mathbb{N} \mid n > 0\}$. The set Ψ is the set of elements of the given security lattice that is $\{\texttt{Lo}, \texttt{Hi}\}$ in our paper.

Our model of a program state consists of a store, a heap, and a set of security upper bounds of values. A Store is defined as an infinite function from $\text{Vars} \cup \{\theta\}$ to $\mathbb{N}$ and is denoted by s. In other words, the domain of s is infinite. A Heap is defined as a finite function from Locs to $\mathbb{N}$ and is denoted by h. A Bound –denoted by b– is defined as an infinite function from $\text{Vars} \cup \text{Locs} \cup \{\theta, \kappa\}$ to Ψ where κ is a special variable that is used to track the security upper bound of the current execution environment. We also refer to κ as the context of the execution. A State is defined by a tuple of Store $\times$ Heap $\times$ Bound and is denoted by σ. We define $s|_V$ to be $s(x)$ for all $x \in V$ and undefined otherwise. We define $b|_V$ in a similar way. We define $\overrightarrow{x}$ to be the sequence $x_0, \ldots, x_k$ for some k and we sometimes use it to denote the set $\{x_i \mid 0 \leq i \leq k\}$ as well.

We define the operational semantics of e to be a function of type $\mathcal{P}(e) \times$ State $\rightarrow$ State. $[\![e]\!]((s, h, b))$ evaluates the expression e for the given state (s, h, b). We also define the semantics of the binary operators as follow. $[\![n_1 + n_2]\!]$ is the addition of n_1 and n_2; $[\![n_1 - n_2]\!]$ is the subtraction of n_2 from n_1; and $[\![n_1 = n_2]\!]$ and $[\![n_1 < n_2]\!]$ are 1 if $n_1 = n_2$ and $n_1 < n_2$ are true respectively and 0 otherwise. We take F to be the set of functions in P.

Definition 3. *The function* $\text{next}(h, k)$ *of type* $Heap \times \mathbb{N} \to \mathbb{N}$ *is defined to be the smallest* n *such that* $n > 0$, $n + 0, \ldots, n + k \notin Dom(h)$.

Below we define the security-aware operational semantics of our programming language.

Definition 4. *We define the security-aware operational semantics of our programming language as below.*

$$\frac{}{\langle c, (s, h, b)\rangle \longrightarrow_\kappa (s[\theta := c], h, b[\theta := \kappa])}\ \text{(CONST)}$$

$$\frac{}{\langle v, (s, h, b)\rangle \longrightarrow_\kappa (s[\theta := s(v)], h, b[\theta := b(v) \sqcup \kappa])}\ \text{(VAR)}$$

$$\frac{n = \text{next}(h, k) \quad b' = b[\theta := \kappa, n + 0 := \kappa \sqcup b(v_0), \ldots, n + k := \kappa \sqcup b(v_k)]}{\begin{array}{c}\langle \text{new } C(v_0, \ldots, c_k), (s, h, b)\rangle \longrightarrow_\kappa \\ (s[\theta := n], h[n + 0 := s(v_0), \ldots, n + k := s(v_k)], b')\end{array}}\ \text{(NEW)}$$

$$\frac{s(v) + i \in \text{Dom}(h)}{\langle v.i, (s, h, b)\rangle \longrightarrow_\kappa (s[\theta := h(s(v) + i)], h, b[\theta := b(s(v) + i) \sqcup b(v) \sqcup \kappa])}\ \text{(READ)}$$

$$\frac{\langle e_1, (s, h, b)\rangle \longrightarrow_\kappa (s_1, h, b_1) \quad \langle e_2, (s, h, b)\rangle \longrightarrow_\kappa (s_2, h, b_2)}{\langle e_1 \diamond e_2, (s, h, b)\rangle \longrightarrow_\kappa (s[\theta := [\![s_1(\theta) \diamond s_2(\theta)]\!]], h, b[\theta := b_1(\theta) \sqcup b_2(\theta)])}\ \text{(BINARY)}$$

$$\frac{\langle e, (s, h, b)\rangle \longrightarrow_\kappa (s', h, b')}{\langle x := e, (s, h, b)\rangle \longrightarrow_\kappa (s[x := s'(\theta)], h, b[x := b'(\theta)])}\ \text{(ASSIGN)}$$

$$\frac{s(v) + i \in \text{Dom}(h) \quad n = s(v) + i}{\langle v.i := v', (s, h, b)\rangle \longrightarrow_\kappa (s, h[n := s(v')], b[n := b(v') \sqcup \kappa])}\ \text{(WRITE)}$$

$$\frac{s(v) \neq 0 \quad \langle e_1, (s, h, b)\rangle \longrightarrow_{\kappa \sqcup b(v)} (s', h', b')}{\langle \text{if } v \text{ then } e_1 \text{ else } e_2, (s, h, b)\rangle \longrightarrow_\kappa (s', h', b')}\ \text{(IF-TRUE)}$$

$$\frac{s(v) = 0 \quad \langle e_2, (s, h, b)\rangle \longrightarrow_{\kappa \sqcup b(v)} (s', h', b')}{\langle \text{if } v \text{ then } e_1 \text{ else } e_2, (s, h, b)\rangle \longrightarrow_\kappa (s', h', b')}\ \text{(IF-FALSE)}$$

$$\frac{\langle e_1, (s, h, b)\rangle \longrightarrow_\kappa (s_1, h_1, b_1) \quad \langle e_2, (s_1, h_1, b_1)\rangle \longrightarrow_\kappa (s', h', b')}{\langle e_1; e_2, (s, h, b)\rangle \longrightarrow_\kappa (s', h', b')}\ \text{(COMP)}$$

$$\frac{\begin{array}{l} f(\overrightarrow{r}, \overrightarrow{u})(\Delta_{\text{pre}})(\Delta_{\text{post}})\{e\} \in F \\ \overrightarrow{l} = \text{FV}(e) - (\overrightarrow{u} \cup \overrightarrow{r}) \qquad \overrightarrow{l}' \text{ fresh} \\ \langle e[\overrightarrow{l} := \overrightarrow{l}'][\overrightarrow{r} := \overrightarrow{v}][\overrightarrow{u} := \overrightarrow{s(y)}], (s, h, b)\rangle \longrightarrow_\kappa (s', h', b') \end{array}}{\langle f(\overrightarrow{y}, \overrightarrow{v}), (s, h, b)\rangle \longrightarrow_\kappa (s', h', b')}\ \text{(RECURSION)}$$

The definition of the operational semantics for our programming language above is an ordinary operational semantics extended with security-awareness. It achieves security-awareness by keeping track of the security upper bounds of the data and propagating it throughout the execution. Thus, it tracks which data

might have been influenced by some high-security data and may eventually lead to a leak. The machine configurations on which the semantics operates consists of a program state and an expression. The set of function declarations F is also taken into consideration, and is implicit in the configuration. It is not made explicit for the sake of brevity. Note that the context, κ, is the standard information flow control construct. It keeps track of the information flow resulting from the control flow of the execution. When an execution enters a branching construct, it elevates the upper bound of the context κ based on the security upper bound of the variable that represents the conditional expression.

Since transformation of the store and the heap due to the execution is straightforward and similar semantics are discussed in [1, 2], we will discuss the part of the semantics that deals with the security upper bounds only.

In the rule (CONST), κ is propagated to the resulting variable θ.

In the rule (VAR), the least upper bound of κ and the variable v is propagated to the resulting variable θ.

In the rule (NEW), *next*(h,k) is used to get the first consecutive k number of unallocated locations in h. For example, in a heap $h_1 = \{(1, 0), (3, 0)\}$, *next*(h_1, 2)=4 and in a heap $h_2 = \{(1, 0), (4, 0)\}$, *next*(h_2, 2)=2. Thus, the semantics are deterministic. The least upper bound of κ and security upper bound of the initializer variables are propagated the corresponding fields and finally κ propagates to the resulting variable θ.

The least upper bound of κ, security upper bound of v, and that of $v.i$ propagates to the resulting variable θ in the rule (READ).

In both the rule (READ) and (WRITE), it is assumed that v is an instance of a class C, which has the declared field i. Hence if the value of v in s is n then $n + i$ is in the domain of both h and b. In other words, value of $v.i$ is $h(s(v) + i)$ and the security upper bound of $v.i$ is $b(s(v) + i)$. Note that unlike the definition of program semantics in [1], we do not have abort.

In the rule (ASSIGN), the least upper bound of ς_κ and upper bounds of all the free variables in e propagates to θ in b' and then it propagates to x. The propagation of κ and security upper bound of v' to $v.i$ is also similar.

In the rule (BINARY), b_1 and b_2 have their own θ where the κ has already been propagated to. The least upper bound of $b_1(\theta)$ and $b_2(\theta)$ is then propagated to the resulting variable θ and thus, κ also propagates to θ.

In the rule (IF-TRUE) and IF-FALSE, κ is raised for their corresponding expression block if the security upper bound of the conditional variable is higher.

In the rule (RECURSION), $\overrightarrow{u}$ and e are respectively the set of parameters and the body of function f. $\overrightarrow{l}$ is the set of local variables (variables other than parameters) in e. $\overrightarrow{l}'$ is a list of fresh variables with the same number of variables as $\overrightarrow{l}$. Note that we do not have global variables. $\overrightarrow{l}$ are substituted by $\overrightarrow{l}'$ in e to avoid name conflicts of newly introduced variables in e with the existing variables of the Store. Since local variables are fresh after substitution, there is no harm in keeping them in the Store even after the execution of e. For example, consider these two functions:

```
f1(p) (p <: Lo) (res <: Lo) { x := p; x }
f2() () {res <: Lo} { x := 1; f1(x) }
```

In both functions above, `x` is local but they might have a name clash with each other when `f1` is unfolded So, we first rename `x` in `f1` to a fresh variable `x'` before calling `f1` from `f2`. Then we have now `f1(p) (p <: Hi) (res <: Hi) {x' := p; x'}`. Hence the names no longer clash with each other. For example, since $(x := p; x)[x := x'][p := x]$ is the same as $x' := x; x'$, and $\langle x' := x; x', (\{(x, 1)\}, \emptyset, \{(x, \texttt{Lo})\})\rangle \longrightarrow (\{(x, 1), (x', 1), (\theta, 1)\}, \emptyset, \{(x, \texttt{Lo}), (x', \texttt{Lo}), (\theta, \texttt{Lo})\})$, we have $\langle f1(x), (\{(x, 1)\}, \emptyset, \{(x <: \texttt{Lo})\})\rangle \longrightarrow (\{(x, 1), (x', 1), (\theta, 1)\}, \emptyset, \{(x <: \texttt{Lo}), (x' <: \texttt{Lo}), (\theta, \texttt{Lo})\})$.

Now we define the semantics of our assertion language. Our assertion language consists of two parts - a separation logic formula and bounds. For the semantics of the separation logic formula, we use the standard semantics as defined in [21, 1]. That is - $[\![A]\!]_{(s,h,b)}$ is true if and only if $[\![A]\!]_{(s,h)}$ is true. Now we define the semantics of a bound formula.

Definition 5. *The meaning of the bound formula is defined below.*

$$\begin{array}{rll} [\![v <: \varsigma]\!]_{(s,h,b)} & \texttt{iff} & b(v) \sqsubseteq \varsigma \\ [\![\theta <: \varsigma]\!]_{(s,h,b)} & \texttt{iff} & b(\theta) \sqsubseteq \varsigma \\ [\![v.i <: \varsigma]\!]_{(s,h,b)} & \texttt{iff} & b(s(v) + i) \sqsubseteq \varsigma \\ [\![\Phi_1 \wedge \Phi_2]\!]_{(s,h,b)} & \texttt{iff} & s, h, b \models \Phi_1 \texttt{ and } s, h, b \models \Phi_2 \end{array}$$

Static verification is done on the entire program P modularly. In particular, for each function $f(\overrightarrow{r}, \overrightarrow{v})(\Delta)(\Delta')\{e\}$, we infer the strongest postcondition from the precondition Δ and expression e. We then check that this strongest postcondition entails the given postcondition Δ'. The strongest postcondition is computed by the inference rule in Definition 13. In particular, algorithm to compute the strongest precondition is a direct implementation of Definition 13.

Now we define the meaning of an entailment in our assertion language.

Definition 6. $A_1 \wedge \Phi_1 \models A_2 \wedge \Phi_2$ *is true if and only if for all* $w <: \varsigma_w$ *in* Φ_2 *there exists* $z <: \varsigma_z$ *in* Φ_1 *such that* $A_1 \models z = w$ *and* $\varsigma_z <: \varsigma_w$.

Here we define the semantics of the asserted program $\{\Delta\}e\{\Delta'\}$. It is inspired from the ordinary separation logic [1]. But one exception is that we do not have the condition on abort.

Definition 7. $\kappa \models \{\Delta\}e\{\Delta'\}$ *is defined to be True if and only if for all* (s, h, b) *and* (s', h', b') *if* $[\![\Delta]\!]_{(s,h,b)} =$ *True and* $\langle e, (s, h, b)\rangle \longrightarrow_\kappa (s', h', b')$ *then* $[\![\Delta']\!]_{(s',h',b')} =$ *True.*

Note that termination of a program is a condition of the above definition. If a program e does not terminate, then there is no (s, h, b) and (s', h', b') such that $\langle e, (s, h, b)\rangle \longrightarrow_\kappa (s', h', b')$ holds. Here we define semantics of a judgement.

Definition 8. $\Gamma, \kappa \vdash \{\Delta\}e\{\Delta'\}$ *is defined to be true when the following holds:* $\kappa \models \{\Delta\}e\{\Delta'\}$ *is True if* $\texttt{Lo} \models \{\Delta_i\}e\{\Delta'_i\}$ *is True for all* $\{\Delta_i\}e\{\Delta'_i\} \in \Gamma$.

In Definition 8, we start the modular verification from low context. Hence, we define judgment based on $\texttt{Lo} \models \{\Delta_i\}e\{\Delta'_i\}$.

Definition 9. $F(\overrightarrow{r}, \overrightarrow{v})(A \wedge \Phi)(A' \wedge \Phi')\{e\}$ *is valid if and only if* $\texttt{Lo} \models \{A \wedge \Phi_1\}e\{A' \wedge \Phi'\}$ *is True,* $(\overrightarrow{r} \cup \overrightarrow{v}) \in \mathrm{FV}(\Phi)$ *and* $(\overrightarrow{r} \cup \{\theta\}) \in \mathrm{FV}(\Phi')$ *where* $\Phi_1 = \Phi \wedge \bigwedge_{x \in \mathrm{FV}(e) - (\overrightarrow{r} \cup \overrightarrow{v})} x <: \texttt{Hi}$.

4 Logical System

In this section we define our logical system.

We write $A[x := e]$ for the formula obtained from A by replacing x by e. We will write the formula $e \mapsto e_1, e_2$ to denote $(e \mapsto e_1) * (e + 1 \mapsto e_2)$. We will also write $\{x_1, \ldots, x_n\} <: \varsigma_x$ to denote $x_1 <: \varsigma_x \wedge \ldots \wedge x_n <: \varsigma_x$.

Definition 10. *Disjunction over our formula,* $(A_1 \wedge \Phi_1) \vee (A_2 \wedge \Phi_2)$*, is defined to be* $(A_1 \vee A_2) \wedge \Phi$ *where* $\forall x(\Phi \models x <: \varsigma \iff \exists \varsigma_1 \varsigma_2(\varsigma = \varsigma_1 \sqcup \varsigma_2 \wedge \Phi_1 \models x <: \varsigma_1 \wedge \Phi_2 \models x <: \varsigma_2))$.

Definition 11. $A \wedge \Phi_1 \circ_v B \wedge \Phi_2$ *is an abbreviation of* $\exists x.(A \wedge \Phi_A)[v := x] \circ B \wedge \Phi_2$ *where* x *is fresh and* $\circ$ *is either* $\wedge$ *or* $*$.

Definition 12. *We define* β *to be a function of type* $\Delta \times E \rightarrow \varsigma$ *as follows.*

$$\beta(A \wedge \Phi, z) = \begin{cases} \varsigma & \Phi \models z <: \varsigma \\ \texttt{Hi} & \text{otherwise} \end{cases}$$

The intuition behind the default security bound being `Hi` is to forbid downgrading the security upper bound of a value. Although we may replace ς with `Lo` in the definition of β, we keep it as ς to allow for extension to an arbitrary lattice of security level. We will discuss our logical system in two parts. First, we will give the inference rules for the language without recursive function. Next, we will provide the inference rules to handle recursive function call.

4.1 System without Recursion

Definition 13. *Our logical system for the language without function call consists of the following inference rules.*

$$[\textsc{Const}]\ \frac{}{\Gamma, \kappa \vdash \{\Delta\}c\{\Delta \wedge_\theta \theta{=}c \wedge \theta <: \kappa\}}$$

$$[\textsc{Var}]\ \frac{}{\Gamma, \kappa \vdash \{\Delta\}v\{\Delta \wedge_\theta \theta{=}v \wedge \theta <: (\beta(\Delta, v) \sqcup \kappa)\}}$$

$$[\textsc{New}]\ \frac{}{\Gamma, \kappa \vdash \begin{array}{c}\{\Delta\} \\ \text{new } C(v_0, \ldots, v_k) \\ \{\Delta *_\theta \theta \mapsto (i_0 : v_0, \ldots, i_k : v_k) \wedge \theta <: \kappa \wedge \bigwedge_{j=0}^{k} \theta.i_j <: (\beta(\Delta, v_j) \sqcup \kappa)\}\end{array}}$$

$$[\textsc{Read}]\quad \frac{\Delta \models x = v \quad \varsigma_\theta = (\beta(\Delta, v) \sqcup \beta(\Delta, x.i) \sqcup \kappa)}{\Gamma, \kappa \vdash \begin{array}{c}\{\Delta * x \mapsto (\ldots, i : w, \ldots)\}\\ v.i \\ \{\Delta * x \mapsto (\ldots, i : w, \ldots) \wedge_\theta \theta = w \wedge \theta <: \varsigma_\theta\}\end{array}}$$

$$[\textsc{Write}]\quad \frac{\Delta \models x = v_1 \quad \varsigma_\theta = (\beta(\Delta, v_1) \sqcup \beta(\Delta, v_2) \sqcup \kappa)}{\Gamma, \kappa \vdash \begin{array}{c}\{\Delta * x \mapsto (\ldots, i : _, \ldots)\}\\ v_1.i := v_2 \\ \{\Delta *_{x.i,\theta} x \mapsto (\ldots, i : v_2, \ldots) \wedge x.i <: \varsigma \wedge \theta = v_2 \wedge \theta <: \varsigma_\theta\}\end{array}}$$

$$[\textsc{Assign}]\quad \frac{\Gamma, \kappa \vdash \{\Delta\} e \{\Delta'\}}{\Gamma, \kappa \vdash \{\Delta\} v := e \{\exists x. \Delta'[v := x][\theta := v] \wedge \theta = v \wedge \theta <: \beta(\Delta', \theta)\}}$$

$$[\textsc{If}]\quad \frac{\Gamma, \kappa \sqcup \beta(\Delta, v) \vdash \{\Delta \wedge v \neq 0\} e_1 \{\Delta_1\} \quad \Gamma, \kappa \sqcup \beta(\Delta, v) \vdash \{\Delta \wedge v = 0\} e_2 \{\Delta_2\}}{\Gamma, \kappa \vdash \{\Delta\} \text{if } v \text{ then } e_1 \text{ else } e_2 \{\Delta_1 \vee \Delta_2\}}$$

$$[\textsc{Binary}]\quad \frac{\begin{array}{c}\Gamma, \kappa \vdash \{\Delta\} e_1 \{\Delta \wedge \theta = v_1 \wedge \theta <: s_1\}\\ \Gamma, \kappa \vdash \{\Delta\} e_2 \{\Delta \wedge \theta = v_2 \wedge \theta <: s_2\}\end{array}}{\Gamma, \kappa \vdash \{\Delta\} e_1 \diamond e_2 \{\Delta \wedge_\theta \theta = v_1 \diamond v_2 \wedge \theta <: (s_1 \sqcup s_2)\}}$$

$$[\textsc{Comp}]\quad \frac{\Gamma, \kappa \vdash \{\Delta\} e_1 \{\Delta_1\} \quad \Gamma, \kappa \vdash \{\Delta_1\} e_2 \{\Delta_2\}}{\Gamma, \kappa \vdash \{\Delta\} e_1; e_2 \{\Delta_2\}}$$

$$[\textsc{Consequence}]\quad \frac{\Gamma, \kappa \vdash \{\Delta_1'\} e \{\Delta_2'\} \quad \Delta_1 \vDash \Delta_1' \quad \Delta_2' \vDash \Delta_2}{\Gamma, \kappa \vdash \{\Delta_1\} e \{\Delta_2\}}$$

The rule [New] captures the flow from κ to the θ and all the newly allocated fields. It also captures the flows from all the initializing variables to the fields. We will explain the latter kind of flow with the following example. We can prove $\texttt{Lo} \vdash \{z <: \texttt{Hi}\} x := \text{new } C(z); x.i_1 \{x \mapsto (i_0 : z) \wedge \theta <: \texttt{Hi}\}$ as follows.

1. By [New]:
   $\texttt{Lo} \vdash \{z <: \texttt{Hi}\} \text{new } C(z) \{z <: \texttt{Hi} * \theta \mapsto (i_0 : z) \wedge \theta <: \texttt{Lo} \wedge \theta.i_0 <: \texttt{Hi}\}$
2. By [Assign]:
   $\texttt{Lo} \vdash \{z <: \texttt{Hi}\} x := \text{new } C(z) \{z <: \texttt{Hi} * x \mapsto (i_0 : z) \wedge x <: \texttt{Lo} \wedge$
   $x.i_0 <: \texttt{Hi} \wedge \theta = x \wedge \theta <: \texttt{Lo}\}$
3. Let Δ be $\{z <: \texttt{Hi} * x \mapsto (i_0 : z) \wedge x <: \texttt{Lo} \wedge x.i_0 <: \texttt{Hi} \wedge \theta = x \wedge \theta <: \texttt{Lo}\}$, by [Read]:
   $\texttt{Lo} \vdash \{\Delta\} x.i_1 \{\Delta \wedge_\theta \theta = z \wedge \theta <: \varsigma_\theta\}$ where $\varsigma_\theta = \beta(\Delta, x) \sqcup \beta(\Delta, x.i_0) \sqcup \kappa = \texttt{Hi}$
4. Lastly, by [Consequence]:
   $\texttt{Lo} \vdash \{z <: \texttt{Hi}\} x := \text{new } C(z); x.i_1 \{x \mapsto (i_0 : z) \wedge \theta <: \texttt{Hi}\}$

Similarly, a dual judgement, $\texttt{Lo} \vdash \{z <: \texttt{Lo}\} x := \text{new } C(z); x.i_1 \{x \mapsto (i_0 : z) \wedge \theta <: \texttt{Lo}\}$, is also provable in our system. In a similar way, the flow from the context to the fields can be explained using the provability of $\texttt{Hi} \vdash \{z <: \texttt{Lo}\} x := \text{new } C(z); x.i_1 \{x \mapsto (i_0 : z) \wedge \theta <: \texttt{Hi}\}$.

To invoke the rule [Read] with the expression $v.i$, it is necessary for the precondition Δ to have the allocation $x \mapsto (\ldots, i : w, \ldots)$ such that $\Delta \models x = v$.

Thus aliasing of the objects is handled using separation logic. This rule captures all the flows from the context κ, the object v and all of its aliases, and the field $x.i$ to the result θ. For example, $\texttt{Lo} \vdash \{z = t \wedge t = w * z \mapsto (b : g) \wedge g <: \texttt{Lo} \wedge z <: \texttt{Hi} \wedge z.b <: \texttt{Lo}\} w.b \{w \mapsto (b : g) \wedge \theta = g \wedge \theta <: \texttt{Hi}\}$ is provable in our system.

In case of unavailablity of such an object x or a field i, the judgement cannot be proved.

The rule [Write] ensures that if the security upper bound of an object is Hi, then the security upper bound of all of its fields also remain Hi. But it is not the same if the security upper bound of the object is Lo. In that case, any of its fields may become upper bounded by Hi if that particular field is mutated in a Hi context or by a Hi value.

The rule [If] is not a conventional one since it has partial disjunction - the separation logic parts are disjuncted whereas the bounds are combined. We combine the bounds by taking least upper bound of the upper bounds of each values. This non-standard interpretation of disjunction is needed since our assertion language does not allow disjunction on security bound formula. Hence, we have to combine the security formula. We explain it using the following simple example.

Suppose we have $\texttt{Lo} \vdash \{p <: \texttt{Lo} \wedge h <: \texttt{Hi} \wedge l <: \texttt{Lo}\} h \{\theta = h \wedge \theta <: \texttt{Hi}\}$ and $\texttt{Lo} \vdash \{p <: \texttt{Lo} \wedge h <: \texttt{Hi} \wedge l <: \texttt{Lo}\} l \{\theta = l \wedge \theta <: \texttt{Lo}\}$.

Then $\texttt{Lo} \vdash \{p <: \texttt{Lo} \wedge h <: \texttt{Hi} \wedge l <: \texttt{Lo}\} \text{if } p \text{ then } h \text{ else } l \{(\theta = h \vee \theta = l) \wedge \theta <: \texttt{Hi}\}$ is provable in our system by Definition 10. Because θ is upper bounded by Lo and Hi in the two former judgements. Since the least upper bound of Lo and Hi is Hi, finally θ is upper bounded by Hi.

Like most other works [11, 28] in the relevant area, our [If] rule also lifts the context to prove the sub-properties of the inner blocks if the upper bound of the condition is higher than the context. This lifted context is used only for the inner blocks of the 'if' expression. For example, suppose we want to prove
$\texttt{Lo} \vdash \{p <: \texttt{Hi} \wedge l <: \texttt{Lo} \wedge l' <: \texttt{Lo}\} \text{if } p \text{ then } l \text{ else } l' \{(\theta = l \vee \theta = l') \wedge \theta <: \texttt{Hi}\}$.
For that we have to prove
$\texttt{Hi} \vdash \{p <: \texttt{Hi} \wedge l <: \texttt{Lo} \wedge l' <: \texttt{Lo}\} l \{\theta = l \wedge \theta <: \texttt{Hi}\}$ and
$\texttt{Hi} \vdash \{p <: \texttt{Hi} \wedge l <: \texttt{Lo} \wedge l' <: \texttt{Lo}\} l' \{\theta = l' \wedge \theta <: \texttt{Hi}\}$.
Indeed they are provable using the rule [Var] and [Consequence]. Thus our [If] rule captures the implicit direct flow.

In the rule [Binary], the value and the upper bound of the result θ of an expression is obtained from θ of the judgements regarding its subexpressions. Note that the flow of the context to θ is captured by all the rules for atomic expressions. By induction, we can say that the flow of the context to θ is captured in the judgements of the subexpressions.

4.2 System with Recursion

Now we will discuss the inference rules to handle recursive functions in a modular way.

Definition 14. *Our logical system for the language with recursive functions consists of the inference rules in Definition 13 extended with the following inference rules.*

$$[\textsc{Recursion}]\ \frac{\begin{array}{c}\Delta \vDash \Delta_f[\overrightarrow{r} := \overrightarrow{z}][\overrightarrow{u} := \overrightarrow{y}]\ \{\Delta_f\}f(\overrightarrow{r},\overrightarrow{u})\{\Delta'_f \wedge \theta <: \varsigma\} \in \Gamma \\ f(\overrightarrow{r},\overrightarrow{u})(\Delta_f)(\Delta'_f)\{e_f\} \qquad m = \{\text{Modified}(e_f)\} \cap \{\overrightarrow{r}\}\end{array}}{\Gamma, \kappa \vdash \{\Delta\}f(\overrightarrow{z},\overrightarrow{y})\{\Delta *_{\theta,\overrightarrow{z}} \Delta'_f[\overrightarrow{r} := \overrightarrow{z}] \wedge (\{\theta\} \cup m) <: (\varsigma \sqcup \kappa)\}}$$

$$[\textsc{Frame}]\ \frac{\Gamma, \kappa \vdash \{\Delta\}e\{\Delta'\}}{\Gamma, \kappa \vdash \{\Delta * \Delta_1\}e\{\Delta' * \Delta_1\}} \qquad (\text{FV}(\Delta_1) \cap \text{FV}(e) = \emptyset)$$

In [Recursion], for the function call expression $f(\overrightarrow{z},\overrightarrow{y})$, we assume that we have a function declaration $f(\overrightarrow{r},\overrightarrow{u})(\Delta_f)(\Delta'_f)\{e_f\}$ and so $\{\Delta_f\}f(\overrightarrow{r},\overrightarrow{u})\{\Delta'_f\} \in \Gamma$. Then we say that the function call is valid for Γ, a context κ, and an specification consists of the precondition Δ and the postconditiom Δ_1 if the following hold:

1. the function with its formal parameters as arguments $f(\overrightarrow{r},\overrightarrow{u})$ is valid for its declared precondition Δ_f and the postcondition Δ'_f and $\Delta_1 = \Delta *_{\theta,\overrightarrow{z}} \Delta'_f[\overrightarrow{r} := \overrightarrow{z}] \wedge \theta <: (\beta(\Delta'_f,\theta) \sqcup \kappa)$; and
2. the precondition Δ implies the declared precondition with its parameters substituted by the arguments $\Delta_f[\overrightarrow{r} := \overrightarrow{z}][\overrightarrow{u} := \overrightarrow{y}]$

The arguments $\overrightarrow{z}$ corresponds to the reference parameters $\overrightarrow{r}$. It is assumed that Δ'_f contains all the necessary information about the effect on the reference variables. So, we first substitute $\overrightarrow{z}$ from Δ by the fresh variables. Substitution of $\overrightarrow{r}$ by $\overrightarrow{z}$ in Δ'_f transforms it as the effects of the function body is local to the calling program. Additionally, we elevate the security bound of the result and modified reference variables to the current context κ. We obtained the over-approximation of modified variable in expression e using Modified(e).

For example, suppose we have the declaration
$f(r,v)(r <: \texttt{Lo} \wedge v <: \texttt{Hi})((r \mapsto (f1:1) \vee r \mapsto (f1:2)) \wedge r <: \texttt{Hi} \wedge \theta <: \texttt{Lo})\{r :=$ if v then new $C(1)$ else new $C(2); 0\}$.
Now we can show that the following judgement is not provable.
$\texttt{Lo} \vdash \{z <: \texttt{Lo} \wedge y <: \texttt{Hi}\}f(z,y)\{z <: \texttt{Lo}\}$.
Because, we have
$\texttt{Lo} \vdash \{z <: \texttt{Lo} \wedge y <: \texttt{Hi}\}f(z,y)\{\exists z'.z' <: \mathit{low} \wedge y <: \texttt{Hi} \wedge (z \mapsto (f1:1) \vee z \mapsto (f1:2)) \wedge z <: \texttt{Hi} \wedge \theta <: \texttt{Lo}\}$
is provable and
$\exists z'.z' <: \texttt{Lo} \wedge y <: \texttt{Hi} \wedge (z \mapsto (f1:1) \vee z \mapsto (f1:2)) \wedge z <: \texttt{Hi} \wedge \theta <: \texttt{Lo} \not\vDash z <: \texttt{Lo}$.
A full example is given below to show a potential security leak where our [Recursion] rule failed to prove. Note that in the source code, we write ~ for reference parameters.

For the ultimate verification, we also assume that all the function declarations are locally provable according to the definition below. Although the two conditions $\text{FV}(\Phi) = \overrightarrow{r} \cup \overrightarrow{v}$ and $\text{FV}(\Phi') = \overrightarrow{r} \cup \{\theta\}$ may seem too strong, preprocessing can be done to ensure that the condition is satisfied. The preprocessing adds $x <: \texttt{Hi}$ to Φ for every $x \in \overrightarrow{r} \cup \overrightarrow{v}$ and $x \notin \Phi$. Similarly, it adds $x <: \texttt{Hi}$ to Φ for every $x \in \overrightarrow{r} \cup \{\theta\}$ and $x \notin \Phi'$. Lastly, the preprocessing removes $x <: \varsigma$ in Φ if $x \notin \overrightarrow{r} \cup \overrightarrow{v}$ as well as removes $x <: \varsigma$ in Φ' if $x \notin \overrightarrow{r} \cup \{\theta\}$.

```
data C{ f1 }
f1 (~r, v) (r<:Lo & v<:Hi) ((r->(f1:1) | r->(f1:2)) & r<:Hi & res<:Lo)
{ r := if v then new C(1) else new C(2); 0 }
f2 (z, y) (z <: Lo & y <: Hi) (res <: Lo)
{ x := f(z, y); z.f1 }
```

Fig. 1. Example of potential leakage due to reference variables.

Definition 15. *A function declaration* $f(\overrightarrow{r}, \overrightarrow{u})(A \wedge \Phi)(A' \wedge \Phi')\{e\}$ *is locally provable if* $\texttt{Lo} \vdash \{A \wedge \Phi\} e_f \{A' \wedge \Phi'\}$ *is provable,* $(\overrightarrow{r} \cup \overrightarrow{v}) = \mathrm{FV}(\Phi)$ *and* $(\overrightarrow{r} \cup \{\theta\}) = \mathrm{FV}(\Phi')$*.*

Note that, the context is considered as `Lo` to verify a declared function locally. But it does not affect the context from which the function is being called. Because the flow from the κ to θ is captured in our [Recursion] rule.

Now we will discuss the intuitions behind one of our design decisions.

Substitution of θ

Now we will discuss the intuition behind $\circ_\theta$ in each of the rules. We substitute the θ using $\wedge_\theta$ for preserving soundness. Consider the program: `x = 1`. In the rule [Assign], we first compute e and the result is expressed through θ. In the example, if 1 has the security bound `Hi`, then we have $\theta = 1 \wedge \theta <: \texttt{Hi}$. Now when we assign e to x, the properties of θ should be transferred to x and it is achieved by substitution as follows: $(\theta = b \wedge \theta <: \texttt{Hi})[\theta := x]$. The result is equivalent to $(x = b \wedge x <: \texttt{Hi})$.

5 Correctness

Theorem 1 (Soundness). *If* $\kappa \vdash \{\Delta\} e \{\Delta'\}$ *is derivable in our inference rules, then* $\kappa \models \{\Delta\} e \{\Delta'\}$ *is True based on Definition 7.*

The proof of the theorem is done via structural induction over e and it is straightforward for the system without recursion. However, note that for (WRITE), we record the security bound of the location $s(v) + i$ in b. Thus, all accesses to the same location will see only a single security bound for the location. This is mimicked in [Write] as a rewriting of $x.i$ for all x such that $\Delta \models x = v_1$.

Although the language does not have aborts, our inference rule is designed in such a way that an abort program cannot be verified. For instance, in the rule [Read], the precondition of $v.i$ should contain $v \mapsto \langle i : _ \rangle$. Hence, it ensures that the program does not abort.

For the system with recursion, to ensure correctness and modularity, we require the function declaration to be well-defined. Since reference variables are replaced by the name of argument, these variables will affect the resulting store s'. On the other hand, since non-reference variables are replaced by the value of argument in the store s, these variables will not affect the resulting store s'.

These are enforced by the condition $\mathrm{FV}(\Phi') = \overrightarrow{r} \cup \{\theta\}$ in Definition 15 which states that the security formula of the postcondition must contain exactly the reference variables and θ.

For $[\![\Delta * A \wedge_{\theta,\overrightarrow{z}} \Phi'[\overrightarrow{r} := \overrightarrow{z}] \wedge \theta <: (\varsigma \sqcup \kappa)]\!]_{(s',h',b')} = \mathrm{True}$ to hold, the condition is necessary. Without the condition, we may have $\overrightarrow{u} \subseteq \mathrm{FV}(\Phi')$ and $\overrightarrow{u} \subseteq \Delta$. Consider the case where $u \in \overrightarrow{u}$ such that $(u <: \mathtt{Lo}) \in \Phi'_f$ and $(u <: \mathtt{Hi}) \in \Delta$. Then we have $[\![s,h,b]\!]_{(u <: \mathtt{Hi} \wedge u <: \mathtt{Lo})}$ which is True iff $b(u) \sqsubseteq \mathtt{Lo}$. But $(u <: \mathtt{Hi}) \in \Delta$ implies $b(u) \subseteq \mathtt{Hi}$ and $u \in \overrightarrow{u}$ implies that $b(u)$ should not change.

Thus, without the condition, soundness is not preserved. The intuition behind the condition is that the postcondition contains all the necessary changes to the state of the caller. Since changes are only through reference variables and θ, these should be in the postcondition.

5.1 Non-Interference

Definition 16 (Observable Equivalence). *Suppose σ_1 and σ_2 are program states such that $\sigma_1 = (s_1, h_1, b_1)$ and $\sigma_2 = (s_2, h_2, b_2)$. We say that they are observably equivalent, written $\sigma_1 \sim \sigma_2$, if:*

- *For all program variables x: $b_1(x) = \mathtt{Lo}$ and $s_1(x) = v$ if and only if $b_2(x) = \mathtt{Lo}$ and $s_2(x) = v$*
- *For all program variables x: $b_1(x) = \mathtt{Hi}$ and $s_1(x) = v_1$ if and only if there exists v_2 such that $b_2(x) = \mathtt{Hi}$ and $s_2(x) = v_2$*
- *For all locations l: $b_1(l) = \mathtt{Lo}$ and $h_1(l) = v$ if and only if $b_2(l) = \mathtt{Lo}$ and $h_2(l) = v$*

Observable equivalence depends on the capability of the attacker. In our case, the attacker is capable of seeing the values of all the variables such that the security bound is upper bounded by Lo. Therefore, any variables x or location l such that $b(x)$ (or respectively $b(l)$) is marked as Lo should produce the same result. On the other hand, any variables marked as Hi can have different values.

Since the operational semantics described in §3.1 uses instrumented semantics, we need to show that these two produce equivalent result in standard semantics without instrumentation. The standard semantics is similar to our instrumented semantics with the exclusion of the Bound b. This property can be stated as the following:

Theorem 2. *Suppose $\langle e, (s,h,b)\rangle \longrightarrow_\kappa (s', h', b')$ in the instrumented semantics. Further suppose that $\Gamma, \kappa \vdash \{\Theta \wedge \Phi\}e\{\Theta' \wedge \Phi'\}$. Then for some s'' and h'', $\langle e, (s,h)\rangle \longrightarrow (s'', h'')$ and $\Gamma \vdash \{\Theta\}e\{\Theta'\}$ in the standard semantics such that $(s', h', b') \sim (s'', h'', b')$.*

Theorem 3. *Suppose $\langle e, (s,h)\rangle \longrightarrow (s', h')$ in the standard semantics. Further suppose that $\Gamma \vdash \{\Theta\}e\{\Theta'\}$. Then for some b, b'', s'', and h'', $\langle e, (s,h,b)\rangle \longrightarrow_\kappa (s'', h'', b'')$ and $\Gamma, \kappa \vdash \{\Theta \wedge \Phi\}e\{\Theta' \wedge \Phi'\}$ in the instrumented semantics such that $(s', h', b'') \sim (s'', h'', b'')$.*

We can then state the non-interference property as the following.

Theorem 4 (Non-Interference). *Suppose* $(s_1, h_1, b_1) \sim (s_2, h_2, b_2)$ *such that* $\langle e, (s_1, h_1, b_1)\rangle \longrightarrow_\kappa (s'_1, h'_1, b'_1)$ *and* $\langle e, (s_2, h_2, b_2)\rangle \longrightarrow_\kappa (s'_2, h'_2, b'_2)$*, then* $(s'_1, h'_1, b'_1) \sim (s'_2, h'_2, b'_2)$*.*

The proofs of these theorems are relatively straightforward by structural induction on e. It relies on the fact that the inference rule [If] merges the security bound of the variables modified in either branch via $\Delta_1 \vee \Delta_2$ and Definition 10. Hence, any modification by one branch will affect the security bound of variables on the other branch after the if statement.

Theorem 4 together with Theorem 2 and 3 ensure that the non-interference property is maintained even in standard semantics. Hence, it shows that the inference rules soundly capture the absence of information leakage.

6 Implementation and Result

We have implemented a basic verifier for our programming language and the associated assertion language. With our verifier, we verified the validity of a variety of examples ranging from simple one line programs to programs containing function calls and object aliasing. Figure 2 shows the categories and number of programs in each category of test programs, along with the number of programs we manually check to contain security leaks that we have tested. To further illustrate our logic, we shall step through the verification process in our logic with some examples in this section.

Category	**Total Programs**	**Insecure Programs**
Constant	12	1
Assignment, Binary Operations	20	5
Objects	8	1
Aliasing	8	2
Non-recursive functions	72	19
Recursive functions	9	1
Total	126	29

Fig. 2. Number and Category of Programs Verified

Here we briefly explain the categories.

Constant. Constant refers to functions that make no change to their arguments and simply either return a constant, or the argument unchanged.

Assignment and Binary Operations. Assignment refers to programs that contains information flow using assignments only. Both implicit and explicit flows are verified. Binary operations involve one of our binary operators of $=$, $\leq$, $+$, and $-$.

Objects. These are the programs that make use of objects within their body, such as creating a new object. This excludes cases where aliasing happens as we consider those separately.

Aliasing. Aliasing happens when more than one different names access the same memory location. In our language, this happens when the same object is assigned to another variable, which may or may not be aliased to other variables.

The following examples demonstrates how our system proves a secured program and disproves an insecure program. Here we will show only high level of verification processes for easy understanding which starts with the context Lo. The first example concerns a non-leaking program that is provable within our system.

```
f (p) (p <: Hi) (res <: Lo) {
```
$\{ p <: \texttt{Hi} \}$
```
c1 := new C(0); c2 := new C(0); c3 := c1;
```
$\{ c3 = c1 * c1 \mapsto (f1 : 0) * c2 \mapsto (f1 : 0) \land p <: \texttt{Hi} \land c1 <: \texttt{Lo} \land$
$c1.f1 <: \texttt{Lo} \land c2 <: \texttt{Lo} \land c2.f1 <: \texttt{Lo} \}$
```
t := if p then 1 else 2
```
$\{ (p \neq 0 \land t = 1 \land c3 = c1 * c1 \mapsto (f1 : 0) * c2 \mapsto (f1 : 0))$
$\lor (p = 0 \land t = 2 \land c3 = c1 * c1 \mapsto (f1 : 0) * c2 \mapsto (f1 : 0)) \land$
$p <: \texttt{Hi} \land c1 <: \texttt{Lo} \land c1.f1 <: \texttt{Lo} \land c2 <: \texttt{Lo} \land c2.f1 <: \texttt{Lo} \land t <: \texttt{Hi} \}$
```
c3.f1 := t;
```
$\{ (p \neq 0 \land t = 1 \land c3 = c1 * c1 \mapsto (f1 : 1) * c2 \mapsto (f1 : 0))$
$\lor (p = 0 \land t = 2 \land c3 = c1 * c1 \mapsto (f1 : 2) * c2 \mapsto (f1 : 0)) \land$
$p <: \texttt{Hi} \land c1 <: \texttt{Lo} \land c1.f1 <: \texttt{Hi} \land c2 <: \texttt{Lo} \land c2.f1 <: \texttt{Lo} \land t <: \texttt{Hi} \}$
```
c2.f1;
```
$\{ (p \neq 0 \land t = 1 \land c3 = c1 \land \theta = 0 * c1 \mapsto (f1 : 1) * c2 \mapsto (f1 : 0))$
$\lor (p = 0 \land t = 2 \land c3 = c1 \land \theta = 0 * c1 \mapsto (f1 : 2) * c2 \mapsto (f1 : 0)) \land$
$p <: \texttt{Hi} \land c1 <: \texttt{Lo} \land c1.f1 <: \texttt{Hi} \land c2 <: \texttt{Lo} \land c2.f1 <: \texttt{Lo} \land t <: \texttt{Hi} \land$
$\theta <: \texttt{Lo} \}$
$\models \theta <: \texttt{Lo}$

In the above verification, the lines above and below of each expression are considered to be its precondition and postcondition respectively. Now consider a similar example with a minor change to the last line, returning `c3.f1` instead of `c2.f1`.

...
$\{ (p \neq 0 \land t = 1 \land c3 = c1 * c1 \mapsto (f1 : 1) * c2 \mapsto (f1 : 0))$
$\lor (p = 0 \land t = 2 \land c3 = c1 * c1 \mapsto (f1 : 2) * c2 \mapsto (f1 : 0)) \land$
$p <: \texttt{Hi} \land c1 <: \texttt{Lo} \land c1.f1 <: \texttt{Hi} \land c2 <: \texttt{Lo} \land c2.f1 <: \texttt{Lo} \land t <: \texttt{Hi} \}$
```
c3.f1;
```
$\{ (p \neq 0 \land t = 1 \land c3 = c1 \land \theta = 0 * c1 \mapsto (f1 : 1) * c2 \mapsto (f1 : 0))$
$\lor (p = 0 \land t = 2 \land c3 = c1 \land \theta = 0 * c1 \mapsto (f1 : 2) * c2 \mapsto (f1 : 0)) \land$
$p <: \texttt{Hi} \land c1 <: \texttt{Lo} \land c1.f1 <: \texttt{Hi} \land c2 <: \texttt{Lo} \land c2.f1 <: \texttt{Lo} \land t <: \texttt{Hi} \land$
$\theta <: \texttt{Lo} \}$
$\not\models \theta <: \texttt{Lo}$

Indeed there is a leak in the above example since there is a direct implicit flow from p to t, and an indirect flow from p to θ where θ is given as Lo in the specification. Now we will show that our logical system does not prove the program as safe. The second analysis is the same as the first except for the last

few lines. For brevity, we present only the last few lines, where the analysis is different.

As can be observed from the analysis above, the program's specification is not provable, and hence there is a potential leak.

Non-recursive functions. Function calls refer to programs that call functions. Because the ability to perform verification of such functions is the core of our modular system, these form the bulk of our examples.

Recursive functions. Recursive function calls refer to programs that contain functions that call themselves during their execution, whether directly (calling itself within its own body) or indirectly (mutual recursion). Figure 3 shows an example of a mutual recursion.

```
f(x) (x <: Hi) (res <: Hi)
{ b := x = 0;
  y := if b then 1 else 0; g(y) }
```

```
g(p) (p <: Hi) (res <: Hi)
{ b := p = 0;
  if b then f(p) else 0 }
```

Fig. 3. Example of mutual recursion.

In the above example, all the parameters are bounded by Hi and the result is also bounded by Hi. It means that the programmer intends to use these functions only to deal with secured information.

Here we rigorously demonstrate how our verification system works for the above mututal recursive functions. From the example above, we have $\Gamma = \{\{x <: \texttt{Hi}\}f(x)\{\theta <: \texttt{Hi}\}, \{p <: \texttt{Hi}\}g(p)\{\theta <: \texttt{Hi}\}\}$. First, we will prove that '`f(x)`' is safe.We will prove:
$\Gamma, \texttt{Lo} \vdash \{x <: \texttt{Hi}\} b := x = 0; y := \text{if } b \text{ then } 1 \text{ else } 0; g(y)\{\theta <: \texttt{Hi}\}$

1. By [Var], [Const], [Binary], and [Consequence]:
   $\Gamma, \texttt{Lo} \vdash \{x <: \texttt{Hi}\} x = 0 \{\theta = (x = 0) \wedge \theta <: \texttt{Hi}\}$
2. By [Assign]:
   $\Gamma, \texttt{Lo} \vdash \{x <: \texttt{Hi}\} b := x = 0 \{b = (x = 0) \wedge b <: \texttt{Hi}\}$
3. Since $\texttt{Lo} \sqcup \texttt{Hi} = \texttt{Hi}$, by [Const] and [Consequence]:
   $\Gamma, \texttt{Hi} \vdash \{b = (x = 0) \wedge b \neq 0 \wedge b <: \texttt{Hi}\} 1 \{b <: \texttt{Hi} \wedge \theta = 1 \wedge \theta <: \texttt{Hi}\}$
   and
   $\Gamma, \texttt{Hi} \vdash \{b = (x = 0) \wedge b = 0 \wedge b <: \texttt{Hi}\} 0 \{b <: \texttt{Hi} \wedge \theta = 0 \wedge \theta <: \texttt{Hi}\}$
4. By [If]:
   $\Gamma, \texttt{Lo} \vdash \{b = (x = 0) \wedge b <: \texttt{Hi}\} \text{if } b \text{ then } 1 \text{ else } 0 \{(\theta = 1 \vee \theta = 0) \wedge b <: \texttt{Hi}$
   $\wedge \theta <: \texttt{Hi}\}$
5. By [Assign]:
   $\Gamma, \texttt{Lo} \vdash \{b = (x = 0) \wedge b <: \texttt{Hi}\} y := \text{if } b \text{ then } 1 \text{ else } 0 \{(y = 1 \vee y = 0) \wedge$
   $b <: \texttt{Hi} \wedge y <: \texttt{Hi}\}$
6. By [Comp]:
   $\Gamma, \texttt{Lo} \vdash \{x <: \texttt{Hi}\} b := x = 0; y := \text{if } b \text{ then } 1 \text{ else } 0 \{(y = 1 \vee y = 0) \wedge y <: \texttt{Hi}\}$
7. Since $y <: \texttt{Hi} \models (p <: \texttt{Hi})[p := y]$, by [Recursion]:
   $\Gamma, \texttt{Lo} \vdash \{(y = 1 \vee y = 0) \wedge y <: \texttt{Hi}\} g(y) \{(y = 1 \vee y = 0) \wedge y <: \texttt{Hi} \wedge \theta <: \texttt{Hi}\}$

8. Lastly, by [Comp] and [Consequence]:
   $\Gamma, \texttt{Lo} \vdash \{x <: \texttt{Hi}\} b := x = 0; y := \text{if } b \text{ then } 1 \text{ else } 0; g(y) \{\theta <: \texttt{Hi}\}$

Similarly, our system can also show that the function '`g(p)`' is safe by proving $\Gamma, \texttt{Lo} \vdash \{p <: \texttt{Hi}\} b := p = 0; \text{if } b \text{ then } f(p) \text{ else } 0 \{\theta <: \texttt{Hi}\}$. Thus, our system concludes that the above example is safe (*i.e.,* there is no security leak).

Our verification system is able to identify all test programs that we manually checked to contain security leaks by showing that the program cannot be proved.

7 Future Work and Conclusion

In this section we will briefly discuss some of the limitations of our system and the implementation. We will also try to discuss possible directions of solving it for the future works. At the end, we will give the conclusion.

7.1 Future Work

Implementation Limitations Our implementation only accepts integer parameters for functions, and cannot take objects as parameters. Also, the main concern of the implementation was to verify the correctness of our logic with respect to flows, and as such, it does not fully handle our fragment of separation logic. Furthermore, objects may only have integer fields.

We would like to extend our implementation to fully handle our fragment of separation logic, and perform more in-depth experiments.

False Positives Because our logic does not fully take into account the semantics of the programming language, there are cases where our system fails to prove a valid program. We explore two such cases here.

```
data C { f1 }
g(p) (p <: Hi) (res <: Lo)
{ c1 := new C(1); b := p = p; c1.f1 := if b then 10 else 9; c1.f1 }
```

Fig. 4. If-else always returns true.

In Figure 4, the condition `p = p` is always true, and hence the value of `c1.f1` is always the same regardless of the value of `p`. Hence `c1.f1` can be bounded by `Lo`. However, in our system, because its value is set in a conditional where the test is bounded by `Hi`, `c1.f1` is always bounded by `Hi`, and this causes the verification to fail, despite the fact that it leaks no information.

To handle such issues, one direction is to have two special forms of [If] rule as follows.

[IF-TRUE]
$$\frac{\Delta \vdash v \neq 0 \quad \Gamma, \kappa \vdash \{\Delta\}e_1\{\Delta_1\}}{\Gamma, \kappa \vdash \{\Delta\}\text{if } v \text{ then } e_1 \text{ else } e_2\{\Delta_1\}}$$

[IF-FALSE]
$$\frac{\Delta \vdash v = 0 \quad \Gamma, \kappa \vdash \{\Delta\}e_2\{\Delta_2\}}{\Gamma, \kappa \vdash \{\Delta\}\text{if } v \text{ then } e_1 \text{ else } e_2\{\Delta_2\}}$$

The rule [IF-TRUE] and [IF-FALSE] are for constant truth value of the condition. These rules can be used to eliminate some false positives. In the presence of a constant value of the condition, an 'if-then-else' expression is equivalent to one of its blocks. In this case, there is no implicit flow occurs. For example, $v := 1; x :=$ if v then 1 else 0 is equivalent to $v := 1; x := 1$. So, we can say that $\kappa \vdash \{\Delta \wedge v = 1\}$if v then 1 else $0\{\Delta'\}$ is provable if $\kappa \vdash \{\Delta \wedge v = 1\}1\{\Delta'\}$ is provable.

Another false positive is shown in Figure 5. In this example, the if-else expression returns the same value in both of its branches. In such a case, an attacker cannot infer any information about the conditional since the result is always the same. However, our logical system is unable to prove that there is no information flow leak in such a situation.

```
data C { f1 }
f2 (p) (p <: Hi) (res <: Lo)
{ c1 := new C(1); t := if p then c1 else c1; c1.f1 }
```

Fig. 5. False positive - same return value in both branches of if-else.

However, we may add a special rule, which is a variant of the rule [IF], to handle this situation as follows.

[IF2]
$$\frac{\begin{array}{c}\Gamma, \kappa \sqcup \beta(\Delta, v) \vdash \{\Delta\}e_1\{\Delta_1\} \Gamma, \kappa \sqcup \beta(\Delta, v) \vdash \{\Delta\}e_2\{\Delta_2\} \\ \Delta_1 \iff \Delta_2 \qquad\qquad \Gamma, \kappa \vdash \{\Delta\}e_1\{\Delta_3\}\end{array}}{\Gamma, \kappa \vdash \{\Delta\}\text{if } v \text{ then } e_1 \text{ else } e_2\{\Delta_3\}}$$

That said, it can be difficult to prove that $\Delta_1 \iff \Delta_2$; deciding if two logical formulas are equivalent is known to be coNP for Boolean propositional logic.

7.2 Conclusion

We have introduced a novel modular logic-based static IFC system that verifies each function independently. We have also shown the correctness of our system with respect to termination- and timing-insensitive non-interference .

In comparison with other systems, our static IFC system does not suffer the high overhead and dynamic security label propagation of dynamic IFC systems. We chose logic-based IFC system due to the higher precision in comparison with type-based systems as the latter requires the inviolability of the static security label declaration. This results in an inability to verify programs with transient leak.

Additionally, we provide examples on how the system can be used for verification. Currently, our implementation of the verifier is still limited in its functionality. We leave the implementation of the full functionality of our verifier for future work.

References

1. M. F. Al Ameen, and M. Tatsuta, Completeness for Recursive Procedures in Separation Logic, In *Theoretical Computer Science*, 2016.
2. M. F. Al Ameen. 2016. *Completeness of Verification System with Separation Logic for Recursive Procedures*. Ph.D. Dissertation. SOKENDAI, Kanagawa, Japan.
3. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In Proceedings of the 17th IEEE workshop on Computer Security Foundations (CSFW '04). IEEE Computer Society, Washington, DC, USA, 100-.
4. T. Amtoft and A. Banerjee, Information Flow Analysis in Logical Form, In: *Proceedings of Static Analysis: 11th International Symposium, SAS 2004*, Verona, Italy, August 26-28, 2004.
5. T. Amtoft, S. Bandhakavi, and A. Banerjee, A Logic for Information Flow in Object-Oriented Programs, In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, USA, 2006.
6. T. H. Austin and C. Flanagan. 2010. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (PLAS '10). ACM, New York, NY, USA, , Article 3 , 12 pages.
7. T. H. Austin and C. Flanagan. 2012. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '12). ACM, New York, NY, USA, 165-178.
8. L. Bello and E. Bonelli. 2011. On-the-Fly inlining of dynamic dependency monitors for secure information flow. In *Proceedings of the 8th international conference on Formal Aspects of Security and Trust* (FAST'11), Gilles Barthe, Anupam Datta, and Sandro Etalle (Eds.). Springer-Verlag, Berlin, Heidelberg, 55-69.
9. W.N. Chin, C. David, H.H. Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.* 77, 9 (August 2012), 1006-1036.
10. A. Chudnov and D.A. Naumann, "Information Flow Monitor Inlining," *2010 23rd IEEE Computer Security Foundations Symposium, Edinburgh*, 2010, pp. 200-214.
11. D. Costanzo and Z. Shao, A Separation Logic for Enforcing Declarative Information Flow Control Policies, LNCS 8414, pp 179-198, 2014.
12. D.E. Denning and P.J. Denning, In *Certification of programs for secure information flow*. Commun. ACM 20, 7 (July 1977), 504-513.
13. J.A. Goguen and J. Meseguer, Security Policies and Security Models, In: *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1982.
14. S. Hunt and D. Sands. On flow-sensitive security types. In POPL, pages 7990, 2006.
15. C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. 2013. CrowdFlow: Efficient Information Flow Security. In *Proceedings of the 16th International Conference on Information Security - Volume 7807* (ISC 2013), Yvo Desmedt (Ed.), Vol. 7807. Springer-Verlag New York, Inc., New York, NY, USA, 321-337.

16. B.W. Lampson. 1973. A note on the confinement problem. *Commun. ACM* 16, 10 (October 1973), 613-615.
17. R. Milner. 1982. A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
18. A. Nanevski, A. Banerjee, and D. Garg. 2011. Verification of Information Flow and Access Control Policies with Dependent Types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (SP '11). IEEE Computer Society, Washington, DC, USA, 165-179
19. P. Ørbæk and J. Palsberg. 1997. Trust in the λ-calculus. *J. Funct. Program.* 7, 6 (November 1997), 557-591.
20. David Park. 1981. Concurrency and Automata on Infinite Sequences. In Proceedings of the 5th GI-Conference on Theoretical Computer Science, Peter Deussen (Ed.). Springer-Verlag, London, UK, UK, 167-183.
21. J.C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (LICS '02). IEEE Computer Society, Washington, DC, USA, 55-74.
22. A. Russo and A. Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium* (CSF '10). IEEE Computer Society, Washington, DC, USA, 186-199.
23. A. Sabelfeld, A.C. Myers, A.C.: A Model for Delimited Information Release. In: *ISSS* 2004, pp. 174-191. Springer, Heidelberg (2004).
24. A. Sabelfeld and A. C. Myers. 2006. Language-based information-flow security. *IEEE J.Sel. A. Commun.* 21, 1 (September 2006), 5-19.
25. J.F. Santos and T. Rezk, (2014) An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript. In: Cuppens-Boulahia N., Cuppens F., Jajodia S., Abou El Kalam A., Sans T. (eds) ICT Systems Security and Privacy Protection. SEC 2014. IFIP Advances in Information and Communication Technology, vol 428. Springer, Berlin, Heidelberg.
26. M. Tatsuta, W.N. Chin, and M.F. Al Ameen, Completeness of Pointer Program Verification by Separation Logic, In: *Proceeding of 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009)* 179–188.
27. Tachio Terauchi and Alex Aiken. 2005. Secure information flow as a safety problem. In Proceedings of the 12th international conference on Static Analysis (SAS'05), Chris Hankin and Igor Siveroni (Eds.). Springer-Verlag, Berlin, Heidelberg, 352-367.
28. L. Fennell1 and P. Thiemann, LJGS: Gradual Security Types for Object-Oriented Languages, In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, 2016.
29. D. Volpano, C. Irvine, and G. Smith. 1996. A sound type system for secure flow analysis. *J. Comput. Secur.* 4, 2-3 (January 1996), 167-187.
30. D.M. Volpano and G. Smith. 1997. A Type-Based Approach to Program Security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development* (TAPSOFT '97), Michel Bidoit and Max Dauchet (Eds.). Springer-Verlag, London, UK, UK, 607-621.
31. Winskel, G. The Formal Semantics of Programming Languages: An Introduction, MIT Press, Cambridge, MA, USA 1993,
32. S. A. Zdancewic. 2002. *Programming Languages for Information Security*. Ph.D. Dissertation. Cornell University, Ithaca, NY, USA. AAI3063751.

On Constructivity of Galois Connections

Francesco Ranzato

Dipartimento di Matematica, University of Padova, Italy

Abstract. Abstract interpretation-based static analyses rely on abstract domains of program properties, such as intervals or congruences for integer variables. Galois connections (GCs) between posets provide the most widespread and useful formal tool for mathematically specifying abstract domains. Darais and Van Horn [2016] put forward a notion of constructive Galois connection for unordered sets (rather than posets), which allows to define abstract domains in a so-called mechanized and calculational proof style and therefore enables the use of proof assistants like Coq and Agda for automatically extracting certified algorithms of static analysis. We show here that constructive GCs are isomorphic, in a mathematical meaning which includes sound abstract functions, to so-called partitioning GCs — an already known class of GCs which allows to cast standard set partitions as an abstract domain. Darais and Van Horn [2016] further provide a notion of constructive Galois connection for posets, which we prove to be mathematically isomorphic to plain GCs. Drawing on these findings, we put forward purely partitioning GCs, a novel class of constructive abstract domains for a mechanized approach to abstract interpretation. We show that this class of abstract domains allows us to represent a set partition in a flexible way while retaining a constructive approach to Galois connections.

1 Introduction

Abstract interpretation [4,5] is probably the most used and successful technique for defining approximations of program semantics (or, more in general, of computing systems) to be used for designing provably sound static program analyzers. Abstract domains play a crucial role in any abstract interpretation, since they encode, both logically for reasoning purposes and practically for implementations, which program properties are computed by a static analysis. Since its beginning [4], one major insight of abstract interpretation is given by the use of Galois connections (GCs) for defining abstract domains. A specification of an abstract domain D through a Galois connection prescribes that: (1) both concrete and abstract domains, C and D, are partially ordered, and typically they give rise to complete lattices; (2) concrete and abstract domains are related by a pair of so-called abstraction $\alpha : C \to D$ and concretization $\gamma : D \to C$ maps; (3) α and γ give rise to an adjunction relation: $\alpha(c) \leq_D d \Leftrightarrow c \leq_C \gamma(d)$. GCs carry both advantages and drawbacks. One major benefit of GCs is the so-called calculational style for defining abstract operations [2,17]. If $f : C \to C$ is any concrete operation involved by some semantic definition (e.g., integer addition or multiplication) then a corresponding correct approximation on A is defined by $\alpha \circ f \circ \gamma : A \to A$, which turns out to be the best possible approximation of f on the abstract domain A and, as envisioned by Cousot [2], allows to systematically derive abstract operations

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 452–473, 2018.
https://doi.org/10.1007/978-3-319-73721-8_21

in a correct-by-design manner. On the negative side, GCs have two main weaknesses. First, GCs formalize an ideal situation where each concrete property in C has a unique best abstract approximation in D. Some very useful and largely used abstract domains cannot be defined by a GC, convex polyhedra being a prominent example of abstract domain where no abstraction map can be defined [9]. This problem motivated weaker abstract interpretation frameworks which only need concretization maps [6]. Secondly, it turns out that abstraction maps of GCs cannot be mechanized [18,20], meaning that one cannot use automatic formal proof systems like Coq in order to extract certified algorithms of abstract interpretation, e.g., based on best correct approximations $\alpha \circ f \circ \gamma$, since the existence of an abstraction map would require a non-constructive axiom (see [20, Section 3.3.2]). In other terms, the calculational approach of abstract interpretation cannot be automatized. Notably, Verasco [15,16] (and its precursor described in [1]) is a static analyzer for C which has been formally designed and verified using the Coq proof assistant, and is based on abstract interpretation using concretization maps only. This latter motivation was one starting point of Darais and Van Horn [10] for investigating constructive versions of Galois connections, together with the observation that many useful abstract domains, even if defined by an abstraction map, still would permit a mechanization of their soundness proofs. Also, Darais and Van Horn's approach [10] generalizes 'Galculator' [24], which is a proof assistant based on a given algebra of Galois connections.

Constructive Galois connections (acronym CGCs) [10] stem from the observation that for many commonly used abstract domains[1]: (1) the concrete domain is a powerset (also called collecting) domain $\wp(A)$ of an unordered carrier domain A; (2) the abstraction map α on the powerset $\wp(A)$ is defined as a lifting to the powerset of a basic abstraction function η, called extraction, which is defined just on the carrier domain A and takes values belonging to an unordered abstract domain B, that is, $\eta : A \to B$; (3) the concretization (or interpretation) map $\mu : B \to \wp(A)$ provides a meaning in $\wp(A)$ to basic abstract values ranging in B; (4) the standard α/γ adjunction relation of GCs can be equivalently reformulated in terms of the following correspondence between η and μ: for all $a \in A$ and $b \in B$,

$$a \in \mu(b) \Leftrightarrow \eta(a) = b \qquad \text{(CGC-Corr)}$$

The intuition is similar to GCs: b approximates a set containing a iff b is the abstraction of a. Moreover, CGCs allow to give a soundness condition for pairs of concrete and abstract functions which are defined on the carrier concrete and abstract domains A and B. As a simple example taken from [10, Section 2], the standard parity (toy) abstraction for integer variables can be defined as a CGC as follows. The carrier concrete domain is $\mathbb{Z}$, the unordered parity domain is $\mathbb{P} = \{\textit{even}, \textit{odd}\}$, while abstraction $\text{parity} : \mathbb{Z} \to \mathbb{P}$ and concretization $\mu : \mathbb{P} \to \wp(\mathbb{Z})$ mappings are straightforwardly defined and satisfy (CGC-Corr): $z \in \mu(a) \Leftrightarrow \text{parity}(z) = a$. Also, from a successor concrete operation $\text{succ} : \mathbb{Z} \to \mathbb{Z}$ one can constructively derive a sound abstract successor $\text{succ}_\sharp : \mathbb{P} \to \mathbb{P}$ such that $\text{succ}_\sharp(\textit{even}) = \textit{odd}$ and $\text{succ}_\sharp(\textit{odd}) = \textit{even}$.

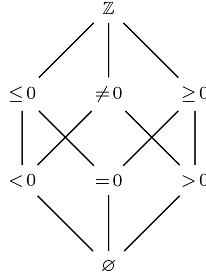
Darais and Van Horn [10] further provide a notion of constructive Galois connection for posets (acronym CGP), where the carrier concrete domain A and the abstract domain

[1] We follow the notation used in [10].

B are posets (rather than unordered sets), and where the above condition (CGC-Corr) is replaced by:

$$a \in \mu(b) \Leftrightarrow \eta(a) \leq_B b \tag{CGP-Corr}$$

This enables a constructive definition for ordered abstract domains like the following abstract lattice Sign:



whose partial order relation $\leq_{\text{Sign}}$ encodes an approximation relation between its abstract values and where $\eta : \mathbb{Z} \to \text{Sign}$ and $\mu : \text{Sign} \to \wp(\mathbb{Z})$. Here, $\eta(a)$ provides the sign of $a \in \mathbb{Z}$ in the subset $\{<0, =0, >0\} \subseteq \text{Sign}$, so that $\eta(a) = b$ of (CGC-Corr) is weakened to $\eta(a) \leq_{\text{Sign}} b$.

Contributions. Our initial observation was that CGCs always encode a partition of the concrete carrier set A. As a simple example, for the above parity domain $\mathbb{P}$, the induced partition of the carrier concrete domain $\mathbb{Z}$ obviously consists of two blocks: $\{z \in \mathbb{Z} \mid z \text{ even}\}$ and $\{z \in \mathbb{Z} \mid z \text{ odd}\}$. Conversely, if an abstract domain D of a powerset domain $\wp(A)$ is defined through a standard Galois connection $\mathcal{G}$ and D does not induce an underlying partition of the carrier set A then we observed that the GC $\mathcal{G}$ cannot be equivalently formulated by a CGC. Abstract domains which encode a partition of a given carrier set have been previously studied and formalized as so-called partitioning Galois connections (PGCs) or elementwise set abstractions [3,7,8]. Intuitively, a Galois connection defining a domain D which abstracts a concrete powerset domain $\wp(A)$ is called partitioning [7,22] when D represents a partition $\mathcal{P}$ of the set A, namely when there exists a partition $\mathcal{P}$ of A such that any $\gamma(d) \in \wp(A)$ is a union of blocks of $\mathcal{P}$. For example, the GC defining the abstract domain Sign above is partitioning, where the induced partition of $\mathbb{Z}$ consists of the blocks $\{z \in \mathbb{Z} \mid z < 0\}$, $\{0\}$ and $\{z \in \mathbb{Z} \mid z > 0\}$.

Our first contribution shows that CGCs are isomorphic to PGCs in the following precise meaning. We define two invertible transforms $\mathbb{T}_{\text{PGC}}$ and $\mathbb{T}_{\text{CGC}}$ such that: (1) $\mathbb{T}_{\text{PGC}}$ transforms any CGC into a PGC; (2) $\mathbb{T}_{\text{CGC}}$ transforms any PGC into a CGC; (3) the transforms are one the inverse of the other, i.e., $\mathbb{T}_{\text{CGC}} \circ \mathbb{T}_{\text{PGC}} = \text{id} = \mathbb{T}_{\text{PGC}} \circ \mathbb{T}_{\text{CGC}}$. Moreover, this isomorphism includes the soundness of abstract operations, meaning that we extend the transforms $\mathbb{T}_{\text{PGC}}$ and $\mathbb{T}_{\text{CGC}}$ in order to convert a pair $\langle f, f^\sharp \rangle$ of concrete and sound abstract operations on a CGC $\mathcal{C}$ to a pair of concrete and sound abstract operations $\mathbb{T}_{\text{PGC}}(\langle f, f^\sharp \rangle)$ on the PGC $\mathbb{T}_{\text{PGC}}(\mathcal{C})$, and analogously the other way round from PGCs to CGCs.

Secondly, we studied Darais and Van Horn's CGPs, in order to investigate whether they can be similarly characterized as a suitable subclass of Galois connections. We

show that CGPs are mathematically equivalent to plain GCs of a powerdomain, meaning that here we define two transforms $\mathbb{T}_{\mathrm{GC}}$ and $\mathbb{T}_{\mathrm{CGP}}$ that give rise to an isomorphism between standard Galois connections relating an abstract domain B to a powerdomain $\wp^{\downarrow}(A)$ and CGPs of the ordered abstract domain B into the carrier set A. Therefore, it turns out that CGPs do not identify a proper subclass of Galois connections.

It is worth remarking that the above transforms $\mathbb{T}_{\mathrm{CGC}}$ and $\mathbb{T}_{\mathrm{CGP}}$ are nonconstructive, meaning that the definitions of $\mathbb{T}_{\mathrm{CGC}}(\mathcal{G})$ and $\mathbb{T}_{\mathrm{CGP}}(\mathcal{G})$ rely on the abstraction map which determines their input Galois connection $\mathcal{G}$. Nonetheless, these transforms are still useful since they provide a formal definition to be used for manually designing a constructive abstract domain starting from a partitioning Galois connection or any Galois connection of a concrete powerdomain.

Drawing on these results, our third contribution is the definition of a novel class of constructive Galois connections, called *purely constructive* GCs (PCGCs). The basic idea underlying PCGCs is as follows. CGCs essentially represent a partition $\mathcal{P}$ of the concrete carrier domain A encoded through an abstract domain B. We showed that this encoding of $\mathcal{P}$ can also be viewed as an implicit representation for all the possible unions of blocks in $\mathcal{P}$. Hence, this observation can be naturally generalized by allowing to select which unions of blocks of $\mathcal{P}$ to consider in the abstract domain B. In other terms, B may be defined as a partition $\mathcal{P}$ of A together with an explicit choice of some unions of blocks of $\mathcal{P}$, where this selection may range from none to all (where all boils down to CGCs). As an example, consider a sign abstraction like $\mathrm{Sign}^{\neq} \triangleq \mathrm{Sign} \setminus \{\neq 0\}$, where the abstract value $\neq 0$ is taken out from the above abstract lattice Sign. Then, it turns out that $\mathrm{Sign}^{\neq}$ cannot be formalized as a CGC, although $\mathrm{Sign}^{\neq}$ still represents a partition of $\mathbb{Z}$. In fact, $\mathrm{Sign}^{\neq}$ just lacks a representation for the union of the two blocks $\{z \in \mathbb{Z} \mid z < 0\}$ and $\{z \in \mathbb{Z} \mid z > 0\}$, i.e., it precisely lacks the removed abstract value $\neq 0$ which would represent this union. In our setting, $\mathrm{Sign}^{\neq}$ can be defined as a PCGC. More precisely, a PCGC of a poset abstract domain B into an unordered concrete carrier set A is defined by $\eta : A \to B$ and $\mu : B \to \wp(A)$ which satisfy the following two conditions:

$$a \in \mu(\eta(a')) \Leftrightarrow \eta(a) = \eta(a') \qquad \text{(PCGC-Corr}_1\text{)}$$
$$a \in \mu(b) \Leftrightarrow \eta(a) \leq_B b \qquad \text{(PCGC-Corr}_2\text{)}$$

Therefore, (PCGC-Corr$_2$) exactly coincides with (CGP-Corr), while (PCGC-Corr$_1$) is a weakening of (CGC-Corr) because it amounts to (CGC-Corr) restricted to abstract values ranging in $\eta(A)$. Thus, as an example, we have that $\mathrm{Sign}^{\neq}$ is a CGP because (PCGC-Corr$_2$) clearly holds, i.e. $a \in \mu(b) \Leftrightarrow \eta(a) \leq_{\mathrm{Sign}^{\neq}} b$ holds, while $\mathrm{Sign}^{\neq}$ is not a CGC because, e.g., $2 \in \mu(\geq 0)$ while $\eta(2) \neq \geq 0$ so that the condition (CGC-Corr) does not hold. On the other hand, let us remark that the weakening (PCGC-Corr$_1$) instead does hold, so that $\mathrm{Sign}^{\neq}$ turns out to be a PCGC. Thus, PCGCs still represent a partition $\mathcal{P}$ of the concrete carrier domain as CGCs do, while retaining a constructive approach to abstract interpretation and providing a flexible way of representing unions of blocks in $\mathcal{P}$. Also, PCGCs come together with a definition of sound abstract operations and of the notion of completeness commonly used in abstract interpretation.

2 Background

Notation. Let $f : A \to B$, $g : A \to \wp(B)$ and $h : \wp(A) \to B$, $k : A \to C$, where A and B are sets and C is a complete lattice with lub $\vee$. We then use the following definitions:

powerset (or collecting) lifting:	$f^{\diamond} : \wp(A) \to \wp(B)$	$f^{\diamond}(X) \triangleq \{f(x) \mid x \in X\}$
singleton powerset lifting:	$f^{\rhd} : A \to \wp(B)$	$f^{\rhd}(a) \triangleq \{f(b)\}$
domain powerset lifting:	$g^{*} : \wp(A) \to \wp(B)$	$g^{*}(X) \triangleq \cup_{x \in X} g(x)$
singleton lowering:	$h^{\{\cdot\}} : A \to B$	$h^{\{\cdot\}}(a) \triangleq h(\{a\})$
lub domain powerset lifting:	$k^{\vee} : \wp(A) \to C$	$k^{\vee}(X) \triangleq \vee_{a \in X} k(a)$

Somewhere we use $f(X)$ as an alternative notation for $f^{\diamond}(X)$. If $f, f' : A \to C$ and C is a poset then we write $f \sqsubseteq f'$ when for any $a \in A$, $f(a) \leq_C f'(c)$. If A is a poset and $X \subseteq A$ then $\downarrow X \triangleq \{y \in A \mid \exists x \in X. y \leq x\}$, and, in turn, $\wp^{\downarrow}(A) \triangleq \{X \in \wp(A) \mid X = \downarrow X\}$ denotes the downward powerdomain of A, which is a complete lattice when it is ordered by subset inclusion. We use $\downarrow a$ as a shorthand for $\downarrow\{a\}$. Recall that any set A can be viewed as a poset w.r.t. the so-called discrete partial order $\leq_d$: for all $x, y \in A$, $x \leq_d y$ iff $x = y$. Let us also recall that $\mathcal{P} \subseteq \wp(A)$ is a partition of A when: (1) $B \in \mathcal{P} \Rightarrow B \neq \varnothing$; (2) if $B_1, B_2 \in \mathcal{P}$ and $B_1 \neq B_2$ then $B_1 \cap B_2 = \varnothing$; (3) $\cup_{B \in \mathcal{P}} B = A$.

Galois Connections. Recall that $\mathcal{G} = \langle \alpha, C, D, \gamma \rangle$ is a Galois connection (GC) when C and D are posets, $\alpha : C \to D$, $\gamma : D \to C$ and $\alpha(c) \leq_D d \Leftrightarrow c \leq_C \gamma(d)$. By following a standard terminology in abstract interpretation, C and D are called concrete and abstract domains, while α and γ are called abstraction and concretization maps. $\mathcal{G}$ is a disjunctive GC when γ is additive (intuitively, this means that $\mathcal{G}$ is able to represent concrete logical disjunctions with no loss of precision). $\mathcal{G}$ is a Galois insertion (GI) when α is surjective, or, equivalently, γ is injective.

Let us also recall some standard definitions and terminology of abstract interpretation [4,5]. Let $f : C \to C$ and $f_\sharp : D \to D$ be, respectively, concrete and abstract functions. We then have the following definitions:

$\langle f, f_\sharp \rangle_{\mathcal{G}}$ is sound if:	$\alpha \circ f \circ \gamma \sqsubseteq f_\sharp$ (equivalently: $\alpha \circ f \sqsubseteq f_\sharp \circ \alpha$)
$\langle f, f_\sharp \rangle_{\mathcal{G}}$ is optimal if:	$\alpha \circ f \circ \gamma = f_\sharp$
$\langle f, f_\sharp \rangle_{\mathcal{G}}$ is backward complete if:	$\alpha \circ f = f_\sharp \circ \alpha$
$\langle f, f_\sharp \rangle_{\mathcal{G}}$ is forward complete if:	$f \circ \gamma = \gamma \circ f_\sharp$
$\langle f, f_\sharp \rangle_{\mathcal{G}}$ is precise if:	$f = \gamma \circ f_\sharp \circ \alpha$

The abstract function $f_{\mathcal{G}} \triangleq \alpha \circ f \circ \gamma$ is called the best correct approximation (b.c.a.) of f induced by $\mathcal{G}$.

Let $\mathcal{G}_1 = \langle \alpha_1, C, D_1, \gamma_1 \rangle$ and $\mathcal{G}_2 = \langle \alpha_2, C, D_2, \gamma_2 \rangle$ be two GCs with a common concrete domain C. $\mathcal{G}_1$ is more precise than $\mathcal{G}_2$, denoted by $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$, when $\gamma_1 \circ \alpha_1 \sqsubseteq \gamma_2 \circ \alpha_2$; This is the standard definition, where the intuition is that the approximation in D_1 is more precise than in D_2, namely, for any $c \in C$, $\gamma_1(\alpha_1(c)) \leq_C \gamma_2(\alpha_2(c))$. Let us also recall that this happens iff $\gamma_2(\alpha_2(C)) \subseteq \gamma_1(\alpha_1(C))$, i.e., any concrete property

which is precisely represented by D_2 is also precisely represented by D_1. In turn, $\mathcal{G}_1$ and $\mathcal{G}_2$ are called isomorphic when $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ and $\mathcal{G}_2 \sqsubseteq \mathcal{G}_1$, i.e., when $\gamma_1 \circ \alpha_1 = \gamma_2 \circ \alpha_2$ holds. Hence, the intuition is that $\mathcal{G}_1$ and $\mathcal{G}_2$ abstractly encode the same properties of C up to a renaming of the abstract values in D_i. This notion can be shifted to abstract functions as follows: If $f_1^\sharp : D_1 \to D_1$ and $f_2^\sharp : D_2 \to D_2$ are two abstract functions for a common concrete function $f : C \to C$ then $f_1^\sharp$ is called isomorphic to $f_2^\sharp$ when $\gamma_1 \circ f_1^\sharp \circ \alpha_1 = \gamma_2 \circ f_2^\sharp \circ \alpha_2$, that is, the following diagram commutes:

$$\begin{array}{ccc} D_1 & \xrightarrow{f_1^\sharp} & D_1 \\ {\scriptstyle\alpha_1}\uparrow & & \downarrow{\scriptstyle\gamma_1} \\ C & & C \\ {\scriptstyle\alpha_2}\downarrow & & \uparrow{\scriptstyle\gamma_2} \\ D_2 & \xrightarrow{f_2^\sharp} & D_2 \end{array}$$

3 Constructive Galois Connections

Constructive Galois connections (CGCs) have been put forward by Darais and Van Horn [10, Section 3] to feature a full "calculational" reasoning style in defining abstract domains and operations, which can therefore support an automatic mechanization by proof assistants. CGCs are defined by a Galois connection-like correspondence between sets rather than posets: $\langle \eta, A, B, \mu \rangle$ is a CGC when A and B are mere sets related by two functions $\eta : A \to B$ and $\mu : B \to \wp(A)$ which satisfy the following equivalence:

$$a \in \mu(b) \;\Leftrightarrow\; \eta(a) = b \qquad \text{(CGC-Corr)}$$

The intuition is that A is a carrier set of the concrete powerset domain $\wp(A)$, B is an unordered abstract domain, η is a representation function (also called extraction function) for concrete singletons $\{a\}$, while μ is a concretization function, which give rise to a sort of unordered adjunction relation between A and B. CGCs enjoy the following two key properties.

Lemma 3.1 (CGC properties). *Consider a CGC* $\langle \eta, A, B, \mu \rangle$.

(1) $\eta(a_1) = \eta(a_2) \;\Leftrightarrow\; \mu(\eta(a_1)) = \mu(\eta(a_2)) \;\Leftrightarrow\; \mu(\eta(a_1)) \cap \mu(\eta(a_2)) \neq \varnothing$
(2) $\mu(b) = \varnothing \;\Leftrightarrow\; b \notin \eta(A)$

Thus, the main consequence of Lemma 3.1 is that $\{\mu(\eta(a))\}_{a \in A}$ are the blocks of a partition of A. In fact, we have that: $A = \cup_{a \in A} \mu(\eta(a))$; by (2), any block $\mu(\eta(a))$ is nonempty; by (1), if $\mu(\eta(a_1)) \neq \mu(\eta(a_2))$ then $\mu(\eta(a_1)) \cap \mu(\eta(a_2)) = \varnothing$. The abstract values ranging in $B \smallsetminus \eta(A)$ can be viewed as "useless" abstract values, because, by Lemma 3.1 (2), they all represent the empty set. This leads to a notion of *constructive Galois insertion* (CGI) which is the analogue of standard Galois insertions: $\langle \eta, A, B, \mu \rangle$ is called a CGI when it is a CGC and η is surjective.

Example 3.2. Consider the unordered abstract domain $B \triangleq \{-, 0, +, \bot\}$, the extraction function $\eta : \mathbb{Z} \to B$ which encodes the sign of an integer, and $\mu : B \to \wp(\mathbb{Z})$ defined by: $\mu(-) \triangleq \mathbb{Z}_{<0}$, $\mu(0) \triangleq \{0\}$, $\mu(+) \triangleq \mathbb{Z}_{>0}$, $\mu(\bot) \triangleq \varnothing$. Then $\mathcal{C} = \langle \eta, \mathbb{Z}, B, \mu \rangle$ is clearly a CGC. This is not a CGI because $\eta(\mathbb{Z}) \subsetneq B$. Let us notice that here $\{\mu(\eta(z))\}_{z \in \mathbb{Z}}$ gives rise to the partition $\{\mathbb{Z}_{<0}, \{0\}, \mathbb{Z}_{>0}\}$ of $\mathbb{Z}$ and that, accordingly with Lemma 3.1 (2), $\mu(\bot)$ must necessarily be set to $\varnothing$, since $\bot \notin \eta(B)$. □

Darais and Van Horn [10, Section 3.1] also define constructive Galois connections for posets (acronym CGPs) as follows. A tuple $\langle \eta, A, B, \mu \rangle$ is a CGP when $\langle A, \leq_A \rangle$ and $\langle B, \leq_B \rangle$ are posets, $\eta : A \to B$ and $\mu : B \to \wp^{\downarrow}(A)$ are monotone and the following equivalence holds:

$$a \in \mu(b) \;\Leftrightarrow\; \eta(a) \leq_B b \qquad \text{(CGP-Corr)}$$

Hence, in (CGP-Corr) the partial order relation $\leq_B$ replaces the equality relation of (CGC-Corr). We also recall that since A is a poset, we have that $\langle \wp^{\downarrow}(A), \subseteq \rangle$ is a complete lattice. It turns out that CGPs have the following properties.

Lemma 3.3 (CGP properties). *Consider a CGP $\langle \eta, A, B, \mu \rangle$.*

(1) $\eta(a_1) = \eta(a_2) \;\Leftrightarrow\; \mu(\eta(a_1)) = \mu(\eta(a_2))$
(2) $\mu(b) = \varnothing \;\Leftrightarrow \downarrow\! b \cap \eta(A) = \varnothing$
(3) *If B is a complete lattice then $\langle \eta^{\vee}, \wp^{\downarrow}(A), B, \mu \rangle$ is a GC.*
(4) $\mu(B) = \mu(\eta^{\vee}(\wp^{\downarrow}(A)))$

Hence, let us remark that by moving from CGCs to CGPs, properties (1) and (2) of Lemma 3.1 are lost and replaced by the weaker properties (1) and (2) of Lemma 3.3. In particular, we lose the key property of CGCs, namely that $\{\mu(\eta(a))\}_{a \in A}$ is partition of the carrier concrete poset A. Let us see an example of this phenomenon.

Example 3.4. Consider $\mathbb{Z}$ with the discrete partial order, so that $\wp^{\downarrow}(\mathbb{Z}) = \wp(\mathbb{Z})$, and consider the following abstract domain B:

$\top$
|
$+$

Let $\eta : \mathbb{Z} \to B$ be defined by $\eta(x) \triangleq$ **if** $x > 0$ **then** $+$ **else** $\top$ and $\mu : \mathbb{Z} \to \wp(\mathbb{Z})$ be defined by $\mu(+) \triangleq \mathbb{Z}_{>0}$ and $\mu(\top) \triangleq \mathbb{Z}$. It turns out that $\mathcal{C} = \langle \eta, \mathbb{Z}, B, \mu \rangle$ is not a CGC, because $1 \in \mu(\top)$ while $+ = \eta(1) \neq \top$. Instead, $+ = \eta(1) \leq_B \top$ holds, and indeed $\mathcal{C}$ turns out to be a CGP. Notice that here $\{\mu(\eta(z)) \mid z \in \mathbb{Z}\} = \{\mathbb{Z}_{>0}, \mathbb{Z}\}$ does not give rise to a partition of $\mathbb{Z}$. □

3.1 Comparing CGCs

In the following we will need to compare CGCs having a common concrete carrier set.

Definition 3.5 (Comparison of CGCs). Let $\mathcal{C}_1 = \langle \eta_1, A, B_1, \mu_1 \rangle$ and $\mathcal{C}_2 = \langle \eta_2, A, B_2, \mu_2 \rangle$ be CGCs. Then, $\mathcal{C}_1$ *more precise than* $\mathcal{C}_2$ (or, $\mathcal{C}_2$ *more abstract than* $\mathcal{C}_1$), denoted by $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$, when:

(1) $\mu_1 \circ \eta_1 \sqsubseteq \mu_2 \circ \eta_2$;
(2) $\eta_1(A) = B_1 \Rightarrow \eta_2(A) = B_2$.

Also, $\mathcal{C}_1$ and $\mathcal{C}_2$ are *isomorphic*, denoted by $\mathcal{C}_1 \cong \mathcal{C}_2$, when $\mathcal{C}_1 \sqsubseteq \mathcal{C}_2$ and $\mathcal{C}_2 \sqsubseteq \mathcal{C}_1$. □

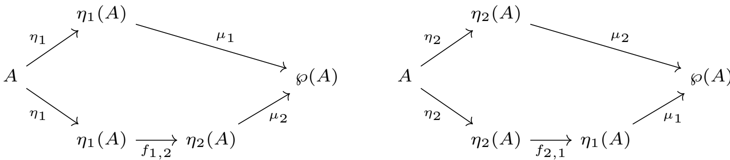
Condition (1) is analogous to GCs and formalizes the intuition that B_1 is a more precise abstract domain than B_2. However, this is not enough for CGCs, because, by Lemma 3.1 (2), if $\eta_2(A) \subsetneq B_2$ then there exists some $b_2 \notin \eta_2(A)$ such that $\mu_2(b_2) = \varnothing$, meaning that B_2 is able to represent the empty property, so that this must also hold for B_1. This is exactly stated by condition (2), which therefore allows us to provide the right comparison relation for CGCs. We also define a *nonempty comparison* relation $\sqsubseteq_\varnothing$ that does not take into account possible empty properties in $\mu(B_i)$: $\mathcal{C}_1 \sqsubseteq_\varnothing \mathcal{C}_2$ when just $\mu_1 \circ \eta_1 \sqsubseteq \mu_2 \circ \eta_2$ holds. In turn, we have nonempty isomorphism: $\mathcal{C}_1 \cong_\varnothing \mathcal{C}_2$ when $\mathcal{C}_1 \sqsubseteq_\varnothing \mathcal{C}_2$ and $\mathcal{C}_2 \sqsubseteq_\varnothing \mathcal{C}_1$.

Lemma 3.6.
(1) $\mathcal{C}_1 \cong \mathcal{C}_2$ *iff* $\mu_1(B_1) = \mu_2(B_2)$.
(2) $\mathcal{C}_1 \cong_\varnothing \mathcal{C}_2$ *iff* $\mu_1(B_1) \smallsetminus \{\varnothing\} = \mu_2(B_2) \smallsetminus \{\varnothing\}$.

Hence, the intuition of the isomorphism relation is the same of Galois connections, as defined in Section 2: two CGCs are isomorphic when they exactly represent the same abstraction of the concrete domain $\wp(A)$ up to a renaming of abstract values. This notion of isomorphism is also justified by the following result, where $f_{1,2}$ and $f_{2,1}$ play the role of renaming functions: $f_{1,2} : \eta_1(A) \to \eta_2(A)$ encodes abstract values in $\eta_1(A)$ as abstract values in $\eta_2(A)$, and conversely for $f_{2,1} : \eta_2(A) \to \eta_1(A)$, where $f_{1,2}$ and $f_{2,1}$ are one the inverse of the other and also commute with the concretizations μ_1 and μ_2.

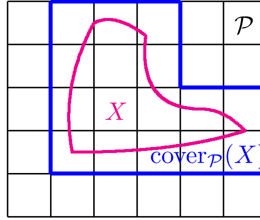
Lemma 3.7 (CGC Isomorphism). *Let* $\mathcal{C}_1 = \langle \eta_1, A, B_1, \mu_1 \rangle$ *and* $\mathcal{C}_2 = \langle \eta_2, A, B_2, \mu_2 \rangle$ *be CGCs. Then,* $\mathcal{C}_1 \cong_\varnothing \mathcal{C}_2$ *iff there exist* $f_{1,2} : \eta_1(A) \to \eta_2(A)$ *and* $f_{2,1} : \eta_2(A) \to \eta_1(A)$ *such that* $f_{1,2} \circ f_{2,1} = \mathrm{id} = f_{2,1} \circ f_{1,2}$, $\mu_1 \circ \eta_1 = \mu_2 \circ f_{1,2} \circ \eta_1$ *and* $\mu_2 \circ \eta_2 = \mu_1 \circ f_{2,1} \circ \eta_2$.

In particular, let us remark that Lemma 3.7 requires that the following two diagrams commute:



4 Partitioning Galois Connections

Partitioning Galois connections/insertions (PGCs/PGIs) have been introduced by Cousot and Cousot as particular examples of Galois connections in a number of articles, where they have been called elementwise set abstractions (or homomorphic abstractions): [7, Section 5], [8, Example 13] and [3, Example 6]. Given a partition $\mathcal{P}$ of a set A, the basic idea is that any subset $X \in \wp(A)$ is over-approximated by the unique minimal cover of X through blocks in $\mathcal{P}$, denoted by $\mathrm{cover}_{\mathcal{P}}(X)$ and depicted in the following picture:



The definition of PGCs given here has been studied and used in [21,22,23] for generalizing strong preservation of temporal logics in model checking. Let us consider a GC $\mathcal{G} = \langle \alpha, \wp(A)_{\subseteq}, D_{\leq}, \gamma \rangle$, where A is any unordered carrier set and D is a complete lattice. Let us remark that, as a consequence of the properties of GCs, D must necessarily be a complete lattice (rather than a mere poset). If $\mathrm{prt}(\mathcal{G}) \triangleq \{\gamma(\alpha(\{a\}))\}_{a \in A}$ then $\mathcal{G}$ is called a *partitioning Galois connection* when:

(1) $\mathrm{prt}(\mathcal{G})$ is a partition of A;
(2) γ is additive, i.e., γ preserves arbitrary lub's.

The main property of a PGC is that any abstract value $d \in D$ represents a union of blocks of the partition $\mathrm{prt}(\mathcal{G})$, namely $\gamma(d) = \cup_{a \in \gamma(d)} \gamma(\alpha(\{a\}))$, and, vice versa, for any set of blocks $\{\gamma(\alpha(\{a\}))\}_{a \in S}$ of the partition $\mathrm{prt}(\mathcal{G})$, for some $S \subseteq A$, there exists $d \in A$ such that $\gamma(d) = \cup_{a \in S} \gamma(\alpha(\{a\}))$. In other terms, the abstract domain D is a representation of all the possible unions of blocks in $\mathrm{prt}(\mathcal{G})$. Alternatively, instead of representing all the possible unions of blocks of a partition, one could equivalently represent no union of blocks at all. This means that the above condition (2), requiring the additivity of the concretization map γ, could be replaced by:

(2′) if $x, y \in D$ and x, y are uncomparable then $\gamma(x \vee_D y) = A$.

In this case, if $\alpha(\{a_1\})$ and $\alpha(\{a_2\})$ represent in D two different blocks then their lub represents no information at all, that is, $\gamma(\alpha(\{a_1, a_2\})) = A$.

Example 4.1. Consider the Sign abstract lattice for sign analysis as depicted in Section 1 and encoded by the GI $\mathcal{S} = \langle \alpha, \wp(\mathbb{Z}), \mathrm{Sign}, \gamma \rangle$, where abstraction and concretization maps are defined as usual. It turns out that $\mathcal{S}$ is a PGC (more precisely, a PGI), where the partition of $\mathbb{Z}$ is given by $\mathrm{prt}(\mathcal{S}) = \{\gamma(\alpha(\{z\})) \subseteq \mathbb{Z} \mid z \in \mathbb{Z}\} = \{\mathbb{Z}_{<0}, \mathbb{Z}_{=0}, \mathbb{Z}_{>0}\}$. □

It turns out that the notion of CGC is equivalent to that of PGC. This equivalence is formalized by two transforms $\mathbb{T}_{\mathrm{PGC}}$ and $\mathbb{T}_{\mathrm{CGC}}$ such that: (1) any CGC $\mathcal{C}$ can be transformed into a PGC $\mathbb{T}_{\mathrm{PGC}}(\mathcal{C})$; (2) any PGC $\mathcal{G}$ can be transformed into a CGC $\mathbb{T}_{\mathrm{CGC}}(\mathcal{G})$; (3) these transforms are one the inverse of the other up to (nonempty) isomorphism, i.e., $\mathbb{T}_{\mathrm{CGC}}(\mathbb{T}_{\mathrm{PGC}}(\mathcal{C})) \cong_{\varnothing} \mathcal{C}$ and $\mathbb{T}_{\mathrm{PGC}}(\mathbb{T}_{\mathrm{CGC}}(\mathcal{G})) \cong \mathcal{G}$.

Theorem 4.2 (CGC-PGC Equivalence).
(1) *If* $\mathcal{C} = \langle \eta, A, B, \mu \rangle$ *is a CGC then* $\mathbb{T}_{\mathrm{PGC}}(\mathcal{C}) \triangleq \langle \eta^{\diamond}, \wp(A)_{\subseteq}, \wp(B)_{\subseteq}, \mu^{*} \rangle$ *is a PGC.*
(2) *If* $\mathcal{G} = \langle \alpha, \wp(A)_{\subseteq}, D_{\leq}, \gamma \rangle$ *is a PGC then* $\mathbb{T}_{\mathrm{CGC}}(\mathcal{G}) \triangleq \langle \alpha^{\{\cdot\}}, A, \{\alpha(\{a\}) \mid a \in A\}, \gamma \rangle$ *is a CGC.*
(3) *The transforms* $\mathbb{T}_{\mathrm{PGC}}$ *and* $\mathbb{T}_{\mathrm{CGC}}$ *are one the inverse of the other, up to nonempty isomorphism.*

Let us remark that in Theorem 4.2, according to the definitions in Section 2:

(1) In the PGC $\mathbb{T}_{\text{PGC}}(\mathcal{C}) = \langle \eta^{\diamond}, \wp(A)_{\subseteq}, \wp(B)_{\subseteq}, \mu^{*}\rangle$, we have that for any $X \in \wp(A)$ and $Y \in \wp(B)$: $\eta^{\diamond}(X) = \{\eta(x) \mid x \in X\} \in \wp(B)$ and $\mu^{*}(Y) = \cup_{y \in Y}\mu(y) \in \wp(A)$.
(2) In the CGC $\mathbb{T}_{\text{CGC}}(\mathcal{G}) = \langle \alpha^{\{\cdot\}}, A, \{\alpha(\{a\}) \mid a \in A\}, \gamma\rangle$, we have that: $\alpha^{\{\cdot\}} : A \to \{\alpha(\{a\}) \mid a \in A\}$ and $\gamma : \{\alpha(\{a\}) \mid a \in A\} \to \wp(A)$, where $\alpha^{\{\cdot\}}(a) = \alpha(\{a\})$ and $\gamma(\alpha(\{a\})) \in \wp(A)$.

Example 4.3. Let us consider the PGC $\mathcal{S} = \langle \alpha, \wp(\mathbb{Z}), \text{Sign}, \gamma\rangle$ of Example 4.1. Then, $\mathbb{T}_{\text{CGC}}(\mathcal{S})$ provides a CGC which is nonempty isomorphic to the CGC $\mathcal{C} = \langle \eta, \mathbb{Z}, B, \mu\rangle$ of Example 3.2: indeed, these two CGCs only differ for the element $\perp \in B$ whose meaning is $\varnothing = \mu(\perp)$. Conversely, $\mathbb{T}_{\text{PGC}}(\mathcal{C})$ is a PGC which is isomorphic to $\mathcal{S}$. In fact, the abstract domain of $\mathbb{T}_{\text{PGC}}(\mathcal{C})$ is $\wp(B)$, so that, since B includes the "useless" value $\perp$, we obtain a PGC rather than a PGI, because its concretization map μ^{*} is not injective, e.g., $\mu^{*}(\{\perp, +\}) = \mu^{*}(\{\perp\})$. □

Furthermore, it turns out that the transforms of Theorem 4.2 preserve the relative precision relations between CGCs/PGCs as follows.

Corollary 4.4. *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are CGCs then $\mathcal{C}_1 \sqsubseteq_{\varnothing} \mathcal{C}_2$ iff $\mathbb{T}_{\text{PGC}}(\mathcal{C}_1) \sqsubseteq \mathbb{T}_{\text{PGC}}(\mathcal{C}_2)$. If $\mathcal{G}_1$ and $\mathcal{G}_2$ are PGCs then $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ iff $\mathbb{T}_{\text{CGC}}(\mathcal{G}_1) \sqsubseteq_{\varnothing} \mathbb{T}_{\text{CGC}}(\mathcal{G}_2)$.*

As a consequence, one can define the lattice of all CGCs having a common concrete carrier set, ordered w.r.t. their relative precision up to nonempty isomorphism $\sqsubseteq_{\varnothing}$, which turns out to be order-theoretically isomorphic to the standard lattice of partitioning abstract domains [23, Theorem 3.2].

Let us mention that [10] also puts forward a notion of *Kleisli* Galois connection (KGC) between posets, which relies on a "monadic" notion of abstraction/concretization maps. Actually, this class of constructive abstract domains is shown to be equivalent to CGCs (cf. [10, Section 6]), where this isomorphism includes the notions of soundness and optimality for abstract functions. Hence, we do not need to replicate our isomorphism between KGCs and PGCs, which comes as a consequence.

CGCs as Least Disjunctive Bases. Given a CGC $\mathcal{C} = \langle \eta, A, B, \mu\rangle$, Theorem 4.2 shows that $\mathbb{T}_{\text{PGC}}(\mathcal{C}) = \langle \eta^{\diamond}, \wp(A)_{\subseteq}, \wp(B)_{\subseteq}, \mu^{*}\rangle$ is a PGC. Let us observe here that $\{\{x\} \mid x \in B\}$ is the set of join-irreducible elements of the complete lattice $\langle \wp(B), \subseteq\rangle$. Recall that an element x of a complete lattice C is join-irreducible when, for any $S \subseteq C$, $x = \vee S \Rightarrow x \in S$, namely when any element $x \in C$ can never be represented as a lub of a subset $S \subseteq C$ not containing x. In abstract interpretation terms (see [11]), this observation means that the set $\{\{x\} \mid x \in B\}$ can be viewed as the so-called *least disjunctive basis* of the partitioning abstract domain $\wp(B)_{\subseteq}$. Least disjunctive bases have been introduced in [11] as an inverse operation to the well-known disjunctive completion of abstract domains [5]. Given an abstract domain D, its least disjunctive basis is defined to be the most abstract domain which has the same disjunctive completion as D. Hence, the least disjunctive basis of D reveals and therefore removes all the disjunctive information inside D by keeping only the information which cannot be reconstructed through logical disjunction. It turns out that a concrete domain which is a powerset,

as it is the case of $\wp(A)_\subseteq$ in the PGC $\mathbb{T}_{\text{PGC}}(\mathcal{C})$, satisfies the hypotheses of [11, Theorem 4.13], and this latter result ensures that the least disjuctive basis of any abstract domain D exists and is characterized as the closure under arbitrary meets of the join-irreducible elements of D. This result can be therefore applied to the abstract domain $\wp(B)_\subseteq$ of the PGC $\mathbb{T}_{\text{PGC}}(\mathcal{C})$, whose least disjunctive basis is given by the meet-closure of $\{\{x\} \mid x \in B\}$. We observe that this meet-closure of $\{\{x\} \mid x \in B\}$ simply adds $\varnothing$ and B. Hence, in this sense, the role of the abstract domain B in a CGC $\langle \eta, A, B, \mu \rangle$ can also be viewed as least disjunctive basis of a partitioning abstract domain.

Constructive Closure Operators. In abstract interpretation, abstract domains up to renaming of abstract values are encoded by closure operators on the concrete domain, which turn out to be fully isomorphic to Galois connections [5] and allow to reason on abstract domains independently of a specific representation of abstract values. Recall that a map $\rho : C \to C$ is a closure operator when ρ is monotone, idempotent and extensive (i.e., $x \leq_C \rho(x)$). Hence, the isomorphism between CGCs and PGCs given by Theorem 4.2 leads us to a notion of "constructive closure operator".

Given any concrete unordered carrier set A, a map $\varphi : A \to \wp(A)$ is a *constructive closure operator* (CCO) when the following condition holds:

$$x \in \varphi(y) \;\Leftrightarrow\; \varphi(x) = \varphi(y) \qquad \text{(CCO-Corr)}$$

CCOs turn out to be the right notion, since they do not rely on a specific representation of abstract values and are equivalent to CGCs, as shown by the following result.

Corollary 4.5 (CGC-CCO Equivalence).
(1) *If* $\mathcal{C} = \langle \eta, A, B, \mu \rangle$ *is a CGC then* $\mathbb{T}_{\text{CCO}}(\mathcal{C}) \triangleq \mu \circ \eta : A \to \wp(A)$ *is a CCO.*
(2) *If* $\varphi : A \to \wp(A)$ *is a CCO then* $\mathbb{T}_{\text{CGC}}(\varphi) \triangleq \langle \varphi, A, \{\varphi(a) \mid a \in A\}, \mathrm{id} \rangle$ *is a CGC.*
(3) *The transforms* $\mathbb{T}_{\text{CCO}}$ *and* $\mathbb{T}_{\text{CGC}}$ *are one the inverse of the other, up to nonempty isomorphism.*

Example 4.6. Consider the CGC $\mathcal{C} = \langle \eta, \mathbb{Z}, B, \mu \rangle$ of Example 3.2, where the unordered abstract domain is $B = \{-, 0, +, \bot\}$. The corresponding constructive closure operator $\mathbb{T}_{\text{CCO}}(\mathcal{C}) : \mathbb{Z} \to \wp(\mathbb{Z})$ is therefore trivially defined, by Corollary 4.5 (1), as follows:

$$\mathbb{T}_{\text{CCO}}(z) = \begin{cases} \mathbb{Z}_{<0} & \text{if } z < 0 \\ \{0\} & \text{if } z = 0 \\ \mathbb{Z}_{>0} & \text{if } z > 0 \end{cases}$$

Analogously to closure operators for standard Galois connections, this map $\mathbb{T}_{\text{CCO}}(\mathcal{C})$ allows us to encode the approximation of the constructive Galois connection $\mathcal{C}$ independently of the specific representation of the abstract domain B. □

4.1 Characterization of CGPs

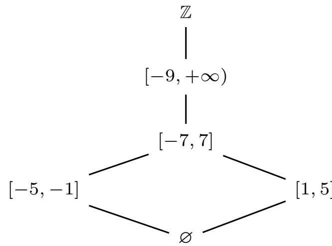
Let us now turn on CGPs. Can this class of constructive abstractions be characterized in terms of some subclass of Galois connections?

Consider a CGP $\langle \eta, A, B, \mu \rangle$, so that the concrete carrier set A is a poset, the abstract domain B is a poset, and the maps $\eta : A \to B$ and $\mu : B \to \wp^{\downarrow}(A)$ are monotone. Here, for our characterization, we additionally need that the abstract domain B is a complete lattice. Following the proof of Theorem 4.2, hence relying on the definition of two CGPs/GCs transforms, we show that the class of CGPs turns out to be isomorphic to the whole class of GCs of the concrete powerdomain $\wp^{\downarrow}(A)$.

Theorem 4.7 (CGP-GC Equivalence).
(1) *If* $\mathcal{C} = \langle \eta, A, B, \mu \rangle$ *is a CGP then* $\mathbb{T}_{\text{GC}}(\mathcal{C}) \triangleq \langle \eta^{\vee}, \wp^{\downarrow}(A)_{\subseteq}, B, \mu \rangle$ *is a GC.*
(2) *If* $\mathcal{G} = \langle \alpha, \wp^{\downarrow}(A)_{\subseteq}, D_{\leq}, \gamma \rangle$ *is a GC then* $\mathbb{T}_{\text{CGP}}(\mathcal{G}) \triangleq \langle \lambda a.\alpha(\downarrow\{a\}), A, D, \gamma \rangle$ *is a CGP.*
(3) *The transforms* $\mathbb{T}_{\text{GC}}$ *and* $\mathbb{T}_{\text{CGP}}$ *are one the inverse of the other, up to isomorphism.*

Otherwise stated, this result shows that the generalization from CGCs to CGPs, which takes care of concrete and abstract carrier sets which are posets, actually provides a constructive characterization of the whole class of Galois connections of the powerdomain $\wp^{\downarrow}(A)$.

Example 4.8. Consider the following lattice D of integer intervals ordered by subset inclusion:



This lattice D gives rise to a Galois insertion $\mathcal{G} = \langle \alpha, \wp(\mathbb{Z}), D, \gamma \rangle$ where $\mathbb{Z}$ is considered with the discrete order, γ is the identity and, for example, we have that $\alpha(\{2\}) = [1, 5]$, $\alpha(\{0\}) = \alpha(\{6\}) = [-7, 7]$, $\alpha(\{10\}) = [-9, +\infty)$, $\alpha(\{-10\}) = \mathbb{Z}$. Let us observe that γ is not additive, because $[-5, 5] = [-5, -1] \cup [1, 5] = \gamma([-5, -1]) \cup \gamma([1, 5]) \subsetneq \gamma([-5, -1] \vee_D [1, 5]) = [-7, 7]$. Hence, this Galois insertion is neither partitioning nor disjunctive.
By Theorem 4.7 (2), it turns out that $\mathbb{T}_{\text{CGP}}(\mathcal{G}) = \langle \lambda z.\alpha(\{z\}), \mathbb{Z}, D, \gamma \rangle$ is a CGP, and this allows us to view D as a constructive abstract domain. Let us remark that this is true even if $\mathcal{G}$ is neither partitioning nor disjunctive.
Let us remark that Theorem 4.7 applies to infinite abstract domains as well. As a simple example, consider the complete lattice $E \triangleq \{[0, n] \mid n \in \mathbb{N}\} \cup \{\mathbb{N}\}$, ordered by subset inclusion, which is an infinite increasing chain of intervals of natural numbers. This complete lattice gives rise to a GI $\mathcal{E} = \langle \alpha, \wp(\mathbb{N}), E, \gamma \rangle$ where $\mathbb{N}$ is discretely ordered and γ is the identity. Here, Theorem 4.7 (2) yields a CGP $\mathbb{T}_{\text{CGP}}(\mathcal{E}) = \langle \eta, \mathbb{N}, E, \text{id} \rangle$ where $\eta(n) = [0, n]$. As a further infinite example, consider the well-known complete lattice of integer intervals Int, which is defined by a GI $\mathcal{I} = \langle \alpha_{\text{Int}}, \wp(\mathbb{N}), \text{Int}, \gamma_{\text{Int}} \rangle$ where $\mathbb{N}$ is discretely ordered [4,5]. Here, Theorem 4.7 (2) yields a CGP $\mathbb{T}_{\text{CGP}}(\mathcal{I}) = \langle \eta_{\text{Int}}, \mathbb{N}, \text{Int}, \gamma_{\text{Int}} \rangle$ where $\eta_{\text{Int}}(n) = [n, n]$. ☐

4.2 On the Meaning of the Isomorphisms

Theorem 4.2 provides an isomorphism between CGCs and partitioning GCs, while Theorem 4.7 yields an isomorphism between CGPs and standard GCs. In particular, Theorem 4.2 (2) shows how a partitioning GC $\mathcal{G}$ can be transformed into an equivalent CGC $\mathbb{T}_{\text{CGC}}(\mathcal{G})$, while Theorem 4.7 (2) establishes how a standard GC $\mathcal{G}$ of a concrete powerdomain can be mapped to an equivalent CGP $\mathbb{T}_{\text{CGP}}(\mathcal{G})$. It should be remarked that the transforms $\mathbb{T}_{\text{CGC}}(\mathcal{G})$ and $\mathbb{T}_{\text{CGP}}(\mathcal{G})$ are nonconstructive, meaning that their definitions rely on the abstraction map α which determines the input Galois connection $\mathcal{G}$. Nevertheless, these transforms are still useful since they provide a precise formal definition which can be used for manually designing a CGC out of a partitioning GC and a CGP out of any GC of a concrete powerdomain, in this latter case thus making possible to define a constructive abstract domain starting from any GC.

5 Soundness of Abstract Operations

Our next step is to transform a pair of sound abstract functions from CGCs to PGCs and vice versa, in order to show that the equivalence between CGCs and PGCs also include soundness of abstract functions. Analogously for optimality. For notational simplicity, we consider unary functions, but the whole approach can be straighforwardly generalized to generic n-ary functions (that indeed we will use in some examples).

Let $\mathcal{C} = \langle \eta, A, B, \mu \rangle$ be a CGC, $f : A \to A$ be a concrete function and $f_\sharp : B \to B$ be a corresponding abstract function. Darais and Van Horn [10] provide four equivalent soudness conditions for the pair $\langle f, f_\sharp \rangle$ w.r.t. $\mathcal{C}$, which are as follows:

$$x \in \mu(y) \;\&\; y' = \eta(f(x)) \;\Rightarrow\; y' = f_\sharp(y) \qquad \text{(CGC-Snd/}\eta\mu)$$
$$x \in \mu(y) \;\&\; x' = f(x) \;\Rightarrow\; x' \in \mu(f_\sharp(y)) \qquad \text{(CGC-Snd/}\mu\mu)$$
$$y = \eta(f(x)) \;\Rightarrow\; y = f_\sharp(\eta(x)) \qquad \text{(CGC-Snd/}\eta\eta)$$
$$x' = f(x) \;\Rightarrow\; x' \in \mu(f_\sharp(\eta(x))) \qquad \text{(CGC-Snd/}\mu\eta)$$

Given two CGCs $\mathcal{C}_i = \langle \eta_i, A, B_i, \mu_i \rangle$, $i = 1, 2$, a concrete function $f : A \to A$ and two corresponding abstract functions $f_i^\sharp : B_i \to B_i$, we extend the notion of CGC isomorphism (given in Section 3.1) to functions as follows: $\langle f, f_1^\sharp \rangle \cong \langle f, f_2^\sharp \rangle$ when (1) f_i is sound for f w.r.t. $\mathcal{C}_i$; (2) $\mu_1 \circ f_1^\sharp \circ \eta_1 = \mu_2 \circ f_2^\sharp \circ \eta_2$. This corresponds to require that the concrete projections of $f_1^\sharp$ and $f_2^\sharp$, which are of type $A \to \wp(A)$, coincide, so that $f_1^\sharp$ and $f_2^\sharp$ can be regarded as being isomorphic.

Let us first consider $\mathbb{T}_{\text{PGC}}$ which transforms a CGC $\mathcal{C}$ into an equivalent partitioning Galois connection $\mathbb{T}_{\text{PGC}}(\mathcal{C}) = \langle \eta^\diamond, \wp(A)_\subseteq, \wp(B)_\subseteq, \mu^* \rangle$. Here, the pair of functions $\langle f, f_\sharp \rangle$ w.r.t. $\mathcal{C}$ is transformed, through the powerset lifting $(\cdot)^\diamond$ of Section 2, into a pair of functions $\mathbb{T}_{\text{PGC}}(\langle f, f_\sharp \rangle)$ w.r.t. $\mathbb{T}_{\text{PGC}}(\mathcal{C})$, that is, $\mathbb{T}_{\text{PGC}}(\langle f, f_\sharp \rangle) \triangleq \langle f^\diamond, f_\sharp^\diamond \rangle$, where $f^\diamond : \wp(A) \to \wp(A)$ and $f_\sharp^\diamond : \wp(B) \to \wp(B)$.

Conversely, let $\mathcal{G} = \langle \alpha, \wp(A)_\subseteq, D_\leq, \gamma \rangle$ be a PGC, so that the abstract domain D represents a partition $\text{prt}(\mathcal{G})$ of A. Here, we need to consider concrete functions on the powerset $\wp(A)$ which are defined as powerset lifting of a mapping $g : A \to A$ on the unordered carrier set A, that is, $g^\diamond : \wp(A) \to \wp(A)$ will be our concrete function. On

the abstract side, a monotone abstract function $g_\sharp : D \to D$ is called *block-preserving* (w.r.t. blocks in $\mathrm{prt}(\mathcal{G})$) when $g_\sharp$ maps (abstract representations of) blocks to (abstract representations of) blocks, namely, when the following condition holds:

$$\forall a \in A. \exists a' \in A.\ g_\sharp(\alpha(\{a\})) = \alpha(\{a'\}).$$

Example 5.1. Consider the PGC (actually PGI) $\mathcal{S} = \langle \alpha, \wp(\mathbb{Z}), \mathrm{Sign}, \gamma \rangle$ of Example 4.1. Similarly to the examples in [10, Section 2], we consider the successor concrete function $succ : \mathbb{Z} \to \mathbb{Z}$ on the concrete carrier domain $\mathbb{Z}$, so that $succ^\diamond : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$. The corresponding best correct approximation $succ_{\mathrm{Sign}} \triangleq \alpha \circ succ^\diamond \circ \gamma$ is as follows:

$$\begin{aligned} succ_{\mathrm{Sign}} = \{ & \varnothing \mapsto \varnothing,\ {<}0 \mapsto {\leq}0,\ {=}0 \mapsto {>}0,\ {>}0 \mapsto {>}0, \\ & {\leq}0 \mapsto \mathbb{Z},\ {\neq}0 \mapsto \mathbb{Z},\ {\geq}0 \mapsto {>}0,\ \mathbb{Z} \mapsto \mathbb{Z}\} \end{aligned}$$

Then, $succ_{\mathrm{Sign}}$ is not block-preserving because $succ_{\mathrm{Sign}}(\alpha(\{-1\})) = {\leq}0$ and there exists no $z \in \mathbb{Z}$ such that $\alpha(\{z\}) = {\leq}0$.
As a further example, consider the concrete square function $sq : \mathbb{Z} \to \mathbb{Z}$, namely, $sq(z) = z^2$, its powerset lifting $sq^\diamond : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, and, in turn, its corresponding best correct approximation $sq_{\mathrm{Sign}} \triangleq \alpha \circ sq^\diamond \circ \gamma$:

$$\begin{aligned} sq_{\mathrm{Sign}} = \{ & \varnothing \mapsto \varnothing,\ {<}0 \mapsto {>}0,\ {=}0 \mapsto {=}0,\ {>}0 \mapsto {>}0, \\ & {\leq}0 \mapsto {\geq}0,\ {\neq}0 \mapsto {>}0,\ {\geq}0 \mapsto {\geq}0,\ \mathbb{Z} \mapsto {\geq}0\} \end{aligned}$$

Here, it turns out that sq_{Sign} is instead block-preserving. □

Lemma 5.2. *If $\mathcal{G}$ is a PGI, $\langle g^\diamond, g_\sharp \rangle$ is sound and $g_\sharp$ is block-preserving then, for any $a \in A$, $g_\sharp(\alpha(\{a\})) = \alpha(\{g(a)\})$ and $g^\diamond(\gamma(\alpha(\{a\}))) \subseteq \gamma(\alpha(\{g(a)\}))$.*

In order to transform a sound pair of functions $\langle g^\diamond, g_\sharp \rangle$ w.r.t. $\mathcal{G}$, where $g_\sharp$ is assumed to be block-preserving, into a pair of functions for $\mathbb{T}_{\mathrm{CGC}}(\mathcal{G}) = \langle \alpha^{\{\cdot\}}, A, \{\alpha(\{a\}) \mid a \in A\}, \gamma \rangle$, we consider:

(i) the concrete carrier function $g : A \to A$
(ii) the restriction $g_\sharp^r$ of the abstract function $g_\sharp$ to abstract representations of blocks, as determined by Lemma 5.2, namely, $g_\sharp^r : \{\alpha(\{a\}) \mid a \in A\} \to \{\alpha(\{a\}) \mid a \in A\}$, with $g_\sharp^r(\alpha(\{a\})) \triangleq \alpha(\{g(a)\})$.

This transform of pair of functions from PGCs to CGCs is denoted by $\mathbb{T}_{\mathrm{CGC}}(\langle g^\diamond, g_\sharp \rangle) \triangleq \langle g, g_\sharp^r \rangle$. It allows us to extend our correspondance between CGCs and PGCs in order to include soundness as follows.

Theorem 5.3.
(1) *Let $\mathcal{C} = \langle \eta, A, B, \mu \rangle$ be a CGC, $f : A \to A$ and $f_\sharp : B \to B$. Then, $\langle f, f_\sharp \rangle$ is sound iff $\mathbb{T}_{\mathrm{PGC}}(\langle f, f_\sharp \rangle)$ is sound w.r.t. $\mathbb{T}_{\mathrm{PGC}}(\mathcal{C})$.*
(2) *Let $\mathcal{G} = \langle \alpha, \wp(A)_\subseteq, D_\leq, \gamma \rangle$ be a PGC, $g^\diamond : \wp(A) \to \wp(A)$, for some $g : A \to A$, and $g_\sharp : D \to D$ be monotone and block-preserving. Then, $\langle g^\diamond, g_\sharp \rangle$ is sound iff $\mathbb{T}_{\mathrm{CGC}}(\langle g^\diamond, g_\sharp \rangle)$ is sound w.r.t. $\mathbb{T}_{\mathrm{CGC}}(\mathcal{G})$.*
(3) *If $\langle f, f_\sharp \rangle$ is sound then $\mathbb{T}_{\mathrm{CGC}}(\mathbb{T}_{\mathrm{PGC}}(\langle f, f_\sharp \rangle)) \cong \langle f, f_\sharp \rangle$. If $\langle g^\diamond, g_\sharp \rangle$ is sound and $g_\sharp$ is block-preserving and additive then $\mathbb{T}_{\mathrm{PGC}}(\mathbb{T}_{\mathrm{CGC}}(\langle g^\diamond, g_\sharp \rangle)) \cong \langle g^\diamond, g_\sharp \rangle$.*

Example 5.4. Consider Example 5.1, where the best correct approximation sq_{Sign} : Sign $\to$ Sign of the concrete square operation $sq^{\diamond} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ is (monotone and) block-preserving. Indeed, the set of (abstract) blocks is $B = \{\alpha(\{z\}) \mid z \in \mathbb{Z}\} = \{<0, =0, >0\}$ and sq_{Sign} maps blocks to blocks. Here, we have that $\mathbb{T}_{\text{CGC}}(\mathcal{S}) = \langle \eta, \mathbb{Z}, B, \mu\rangle$ and $\mathbb{T}_{\text{CGC}}(\langle sq^{\diamond}, sq_{\mathcal{S}}\rangle) = \langle sq, sq^{r}_{\mathcal{S}}\rangle$ where $sq : \mathbb{Z} \to \mathbb{Z}$ and the restriction $sq^{r}_{\mathcal{S}} : B \to B$ is such that $sq^{r}_{\mathcal{S}}(\alpha(\{z\})) = \alpha(\{sq(z)\})$, namely:

$$sq^{r}_{\mathcal{S}} = \{<0 \mapsto >0,\ =0 \mapsto =0,\ >0 \mapsto >0\} \qquad \square$$

5.1 Completeness

As observed in [10], the above four equivalent soundness conditions (CGC-Snd) for CGCs lead to four non-equivalent conditions of completeness for abstract functions, where $\Leftrightarrow$ replaces $\Rightarrow$:

$$x \in \mu(y) \,\&\, y' = \eta(f(x)) \;\Leftrightarrow\; y' = f_{\sharp}(y) \qquad \text{(CGC-Cmp/}\eta\mu\text{)}$$
$$x \in \mu(y) \,\&\, x' = f(x) \;\Leftrightarrow\; x' \in \mu(f_{\sharp}(y)) \qquad \text{(CGC-Cmp/}\mu\mu\text{)}$$
$$y = \eta(f(x)) \;\Leftrightarrow\; y = f_{\sharp}(\eta(x)) \qquad \text{(CGC-Cmp/}\eta\eta\text{)}$$
$$x' = f(x) \;\Leftrightarrow\; x' \in \mu(f_{\sharp}(\eta(x))) \qquad \text{(CGC-Cmp/}\mu\eta\text{)}$$

It turns out that these completeness conditions for a pair $\langle f, f_{\sharp}\rangle$ can be equivalently stated using the standard optimality/completeness/precision conditions for Galois connections, as recalled in Section 2, for the transformed pair $\mathbb{T}_{\text{PGC}}(\langle f, f_{\sharp}\rangle)$.

Lemma 5.5.
(1) $\langle f, f_{\sharp}\rangle$ *satisfies* (CGC-Cmp/$\eta\mu$) *iff* $\mathbb{T}_{\text{PGC}}(\langle f, f_{\sharp}\rangle)$ *is optimal w.r.t.* $\mathbb{T}_{\text{PGC}}(\mathcal{C})$.
(2) $\langle f, f_{\sharp}\rangle$ *satisfies* (CGC-Cmp/$\mu\mu$) *iff* $\mathbb{T}_{\text{PGC}}(\langle f, f_{\sharp}\rangle)$ *is forward complete w.r.t.* $\mathbb{T}_{\text{PGC}}(\mathcal{C})$.
(3) $\langle f, f_{\sharp}\rangle$ *satisfies* (CGC-Cmp/$\eta\eta$) *iff* $\mathbb{T}_{\text{PGC}}(\langle f, f_{\sharp}\rangle)$ *is backward complete w.r.t.* $\mathbb{T}_{\text{PGC}}(\mathcal{C})$.
(4) $\langle f, f_{\sharp}\rangle$ *satisfies* (CGC-Cmp/$\mu\eta$) *iff* $\mathbb{T}_{\text{PGC}}(\langle f, f_{\sharp}\rangle)$ *is precise w.r.t.* $\mathbb{T}_{\text{PGC}}(\mathcal{C})$.

Example 5.6. Consider Example 5.4, namely the CGC $\mathcal{C} = \langle \eta, \mathbb{Z}, B, \mu\rangle$, with $B = \{<0, =0, >0\}$, the concrete square operation $sq : \mathbb{Z} \to \mathbb{Z}$ and the corresponding abstract square operation $sq_{\sharp} : B \to B$

$$sq_{\sharp} = \{<0 \mapsto >0,\ =0 \mapsto =0,\ >0 \mapsto >0\}$$

It turns out that $\langle sq, sq_{\sharp}\rangle$ satisfies (CGC-Cmp/$\eta\mu$), (CGC-Cmp/$\mu\mu$) and (CGC-Cmp/$\eta\eta$) but not (CGC-Cmp/$\mu\eta$). This can be easily checked on the transformed pair of functions $\mathbb{T}_{\text{PGC}}(\langle sq, sq_{\sharp}\rangle) = \langle sq^{\diamond}, sq^{\diamond}_{\sharp}\rangle$ by resorting to Lemma 5.5: in fact, we have that $sq^{\diamond}_{\sharp} : \wp(B) \to \wp(B)$ is clearly both backward and forward complete (and therefore optimal) for $sq^{\diamond} : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, while it is not precise, because $sq^{\diamond} \neq \gamma \circ sq^{\diamond}_{\sharp} \circ \alpha$. $\square$

6 Purely Partitioning Galois Connections

Drawing on the above results, we define a novel class of constructive abstract domains, which we call purely constructive Galois connections (PCGCs). The idea is that PCGCs

generalize CGCs as follows. We have shown that CGCs may be viewed as representing a partition of the concrete carrier domain A through an abstract domain B. We proved that this view of a CGC as a partition also implicitly represents all the possible unions of its blocks. The goal here is to generalize this approach by allowing to choose which unions of blocks to consider in the abstract domain B. Hence, B may be defined as a partition P of A together with an explicit selection of unions of blocks of P, where this selection may range from none to all.

A *purely constructive Galois connection* (PCGC) $\langle \eta, A, B, \mu \rangle$ consists of a concrete unordered carrier set A and of an abstract ordered domain $\langle B, \leq \rangle$ which is required to be a poset, together with two maps $\eta : A \to B$ and $\mu : B \to \wp(A)$ which satisfy the following two conditions:

$$a \in \mu(\eta(a')) \;\Leftrightarrow\; \eta(a) = \eta(a') \qquad \text{(PCGC-Corr}_1\text{)}$$

$$a \in \mu(b) \;\Leftrightarrow\; \eta(a) \leq b \qquad \text{(PCGC-Corr}_2\text{)}$$

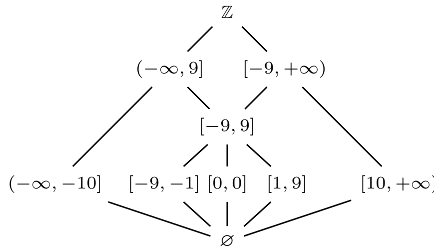
Thus, (PCGC-Corr$_2$) coincides with (CGP-Corr), while the condition (PCGC-Corr$_1$) amounts to (CGC-Corr) restricted to abstract values ranging in $\eta(A)$. PCGCs have the following properties.

Lemma 6.1 (PCGC properties). *Consider a PCGC* $\langle \eta, A, B_{\leq}, \mu \rangle$.
(1) $\eta(a_1) = \eta(a_2) \;\Leftrightarrow\; \mu(\eta(a_1)) = \mu(\eta(a_2)) \;\Leftrightarrow\; \mu(\eta(a_1)) \cap \mu(\eta(a_2)) \neq \varnothing$
(2) $\mu(b) = \varnothing \;\Rightarrow\; b \notin \eta(A)$*, while the viceversa does not hold.*
(3) *If B is a complete lattice then* $\langle \eta^{\vee}, \wp(A)_{\subseteq}, B_{\leq}, \mu \rangle$ *is a GC.*

In particular, let us remark that:

- by Lemma 6.1 (1), which is the same property of Lemma 3.1 (1) for CGCs, we have that $\{\mu(\eta(a))\}_{a \in A}$ still is a partition of A;
- by Lemma 6.1 (2), differently from CGCs (cf. Lemma 3.1 (2)), if $b \notin \eta(A)$ it may happen that $\mu(b) \neq \varnothing$;
- by Lemma 6.1 (3), analogously to CGPs, $\eta^{\vee}$ and μ give rise to a GC, analogously to what happens for CGPs (cf. Lemma 3.3 (3)).

Example 6.2. Consider the following finite lattice B of integer intervals ordered by subset inclusion:

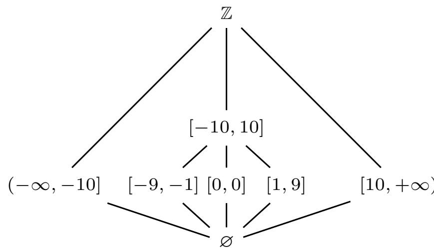


Let $\eta : \mathbb{Z} \to B$ be defined as follows:

$$\eta(x) \triangleq \begin{cases} (-\infty, -10] & \text{if } x \in (-\infty, -10] \\ [-9, -1] & \text{if } x \in [-9, -1] \\ [0, 0] & \text{if } x = 0 \\ [1, 9] & \text{if } x \in [1, 9] \\ [10, +\infty) & \text{if } x \in [10, +\infty) \end{cases}$$

while $\mu : B \to \wp(\mathbb{Z})$ is simply defined as the identity map. Then, it is simple to check that $\mathcal{P} = \langle \eta, \mathbb{Z}, B, \mu \rangle$ is a PCGC. It turns out that $\mathcal{P}$ is not a CGC: in fact, $0 \in \mu([-9, 9])$ while $\eta(0) = [0, 0] \neq [-9, 9]$, thus (CGC-Corr) does not hold. Also, if $\mathbb{Z}$ is considered as a poset w.r.t. the discrete order then $\wp^{\downarrow}(\mathbb{Z}) = \wp(\mathbb{Z})$ and η and μ are monotone functions, so that, since (PCGC-Corr$_2$) holds, $\mathcal{P}$ turns out to be a CGP as well.

Consider now the following lattice B':

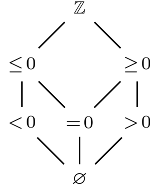


In this case, we have that $\langle \eta, \mathbb{Z}, B', \mu \rangle$ is not a PCGC, because $10 \in \mu([-10, 10])$ but $\eta(10) = [10, +\infty) \not\subseteq [-10, 10]$, so that (PCGC-Corr$_2$) does not hold. The intuition is that while B' still includes a subset which gives rise to a partition of $\mathbb{Z}$, the whole lattice B' cannot be coeherently seen as a partition representation, because $[-10, 10] \in B'$ is not the union (i.e., lub) of the blocks $[-9, -1]$, $[0, 0]$ and $[1, 9]$.

Finally, consider the CGC $\mathcal{C} = \langle \eta, \mathbb{Z}, B, \mu \rangle$ defined in Example 3.4. Then, $\mathcal{C}$ is not a PCGC because $1 \in \mu(\eta(0)) = \mathbb{Z}$ but $+ = \eta(1) \neq \eta(0) = \top$, that is, (PCGC-Corr$_1$) does not hold. □

Similarly to Theorems 4.2 and 4.7, let us now characterize PCGCs as a class of Galois connections. Recall that a GC $\mathcal{G} = \langle \alpha, \wp(A)_{\subseteq}, D_{\leq}, \gamma \rangle$ is a PGC when $\mathrm{prt}(\mathcal{G})$ is a partition of A and γ is additive. By dropping this latter requirement of additivity for γ, we define $\mathcal{G}$ to be a *purely partitioning* Galois connection (PPGC) just when $\mathrm{prt}(\mathcal{G})$ is a partition of A. The terminology "purely partitioning" hints at the property (which is not hard to check) that the disjunctive completion of D indeed yields a partitioning Galois connection.

Example 6.3. Consider the following abstract domain $\mathrm{Sign}^{\neq} \triangleq \mathrm{Sign} \smallsetminus \{\neq 0\}$, already mentioned in Section 1:



Then, $\mathcal{S}^{\neq} = \langle \alpha, \wp(\mathbb{Z}), \mathrm{Sign}^{\neq}, \gamma \rangle$ is not a PCG because γ is not additive (in fact: $\gamma(<0) \cup \gamma(>0) \neq \gamma(<0 \ \vee\ >0) = \gamma(\mathbb{Z})$). However, $\mathrm{prt}(\mathcal{S}^{\neq}) = \{\mathbb{Z}_{<0}, \{0\}, \mathbb{Z}_{>0}\}$ still is a partition of $\mathbb{Z}$, so that $\mathcal{S}^{\neq}$ is a PPCG. □

It turns out that this class of GCs precisely characterize PCGCs as follows.

Theorem 6.4 (PCGC-PPGC Equivalence).
(1) *If $B_{\leq}$ is a complete lattice and $\mathcal{C} = \langle \eta, A, B_{\leq}, \mu \rangle$ is a PCGC then* $\mathbb{T}_{\text{PPGC}}(\mathcal{C}) \triangleq \langle \eta^{\vee}, \wp(A)_{\subseteq}, B_{\leq}, \mu \rangle$ *is a PPGC.*
(2) *If $\mathcal{G} = \langle \alpha, \wp(A)_{\subseteq}, D_{\leq}, \gamma \rangle$ is a PPGC then* $\mathbb{T}_{\text{PCGC}}(\mathcal{G}) \triangleq \langle \alpha^{\{\cdot\}}, A, D_{\leq}, \gamma \rangle$ *is a PCGC.*
(3) *The transforms $\mathbb{T}_{\text{PPGC}}$ and $\mathbb{T}_{\text{PCGC}}$ are one the inverse of the other, up to isomorphism.*

Example 6.5. Let us consider the PCGC $\mathcal{P}$ defined in Example 6.2. Then, $\mathbb{T}_{\text{PPGC}}(\mathcal{P}) = \langle \eta^{\vee}, \wp(\mathbb{Z})_{\subseteq}, B_{\leq}, \mathrm{id} \rangle$ is a purely partitioning GC where the corresponding partition of $\mathbb{Z}$ is

$$P = \{(-\infty, 10], [-9, -1], [0, 0], [1, 9], [10, +\infty)\}$$

and the abstraction map $\eta^{\vee}$ approximates a set of integers $X \in \wp(\mathbb{Z})$ by the least union of blocks of P which belongs to B: for example, $\eta^{\vee}(\{1, 10\}) = [-9, +\infty)$ and $\eta^{\vee}(\{0, 1\}) = [-9, 9]$. □

CGCs as PCGCs as CGPs. Let us show that any CGC is indeed a PCGC, which, in turn, is a CGP. Let $\langle \eta, A, B, \mu \rangle$ be a CGC. Firstly, it is enough to consider B as a poset for the discrete partial order $\leq_d$, since this makes $\langle \eta, A, B_{\leq_d}, \mu \rangle$ a PCGC. In fact: (1) $a \in \mu(\eta(a'))$ iff, by (CGC-Corr), $\eta(a) = \eta(a')$; (2) if $b \in \eta(A)$ then $b = \eta(a')$, for some a', so that, by (CGC-Corr), $a \in \mu(b) \Leftrightarrow \eta(a) = b$, while if $b \notin \eta(A)$, then, by Lemma 3.1 (2), $\mu(b) = \varnothing$. Secondly, any PCGC $\langle \eta, A, B_{\leq_B}, \mu \rangle$ can be viewed as a CGP simply by making the concrete unordered carrier set A a poset for the discrete order $\leq_d$, so that $\wp^{\downarrow}(A) = \wp(A)$, and $\eta : A \to B$ becomes trivially monotone as well as $\mu : B \to \wp(A)$: in fact, if $b_1 \leq_B b_2$ and $a \in \mu(b_1)$ then $\eta(a) \leq_B b_1 \leq_B b_2$, so that $a \in \mu(b_2)$, namely $\mu(b_1) \subseteq \mu(b_2)$.

6.1 Soundness of Abstract Operations

Let $\mathcal{C} = \langle \eta, A, B_{\leq}, \mu \rangle$ be a PCGC and $f : A \to A$ be a concrete function. By relying on Theorem 6.4 (1), we are able to define the best correct approximation of the lifted function $f^{\diamond} : \wp(A) \to \wp(A)$ w.r.t. the PPGC $\langle \eta^{\vee}, \wp(A)_{\subseteq}, B_{\leq}, \mu \rangle = \mathbb{T}_{\text{PPGC}}(\mathcal{C})$. This b.c.a. is denoted by $f_{\mathcal{C}} : B \to B$ and is therefore defined by $f_{\mathcal{C}} \triangleq \eta^{\vee} \circ f^{\diamond} \circ \mu$, so that:

$$f_{\mathcal{C}}(b) = \vee\{\eta(f(a)) \mid a \in \mu(b)\}.$$

Hence, given an abstract function $f_\sharp : B \to B$, this b.c.a. suggests to define $\langle f, f^\sharp \rangle$ to be sound for the PCGC $\mathcal{C}$ when $f_\sharp$ is less precise than the b.c.a., that is, when for any $b \in B$, $f_{\mathcal{C}}(b) \leq f_\sharp(b)$. It is easy to check that this latter condition turns out to be equivalent to the following definition: $\langle f, f^\sharp \rangle$ is sound w.r.t. $\mathcal{C}$ when

$$\eta(a) \leq b \;\Rightarrow\; \eta(f(a)) \leq f_\sharp(b) \qquad \text{(PCGC-Snd)}$$

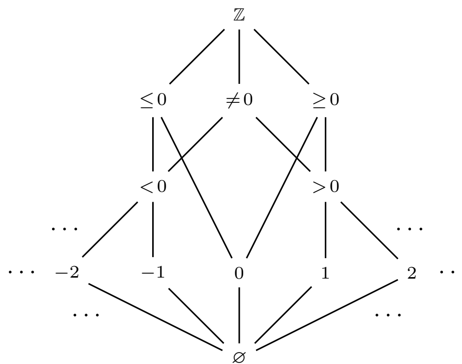
It is then easy to transform a sound pair of concrete/abstract functions $\langle f, f_\sharp \rangle$ for a PCGC $\mathcal{C}$ into a pair $\mathbb{T}_{\text{PPGC}}(\langle f, f_\sharp \rangle) \triangleq \langle f^\diamond, f_\sharp \rangle$ for the corresponding PPGC $\mathbb{T}_{\text{PPGC}}(\mathcal{C}) = \langle \eta^\vee, \wp(A)_\subseteq, B_\leq, \mu \rangle$. Conversely, if $\mathcal{D} = \langle \alpha, \wp(A)_\subseteq, D_\leq, \gamma \rangle$ is a PPGC and $\langle g^\diamond, g_\sharp \rangle$ is a sound pair for $\mathcal{D}$, where $g^\diamond : \wp(A) \to \wp(A)$ for some $g : A \to A$, then $\langle g^\diamond, g_\sharp \rangle$ is transformed into $\mathbb{T}_{\text{PCGC}}(\langle g^\diamond, g_\sharp \rangle) \triangleq \langle g, g_\sharp \rangle$ relatively to the PCGC $\mathbb{T}_{\text{PCGC}}(\mathcal{D})$. Hence, an equivalence result analogous to Theorem 5.3 can then be proved.

Theorem 6.6.
(1) *Let* $\mathcal{C} = \langle \eta, A, B_\leq, \mu \rangle$ *be a PCGC, with* B *complete lattice,* $f : A \to A$ *and* $f_\sharp : B \to B$. *Then,* $\langle f, f_\sharp \rangle$ *is sound iff* $\mathbb{T}_{\text{PPGC}}(\langle f, f_\sharp \rangle)$ *is sound w.r.t.* $\mathbb{T}_{\text{PPGC}}(\mathcal{C})$.
(2) *Let* $\mathcal{D} = \langle \alpha, \wp(A)_\subseteq, D_\leq, \gamma \rangle$ *be a PPGC,* $g^\diamond : \wp(A) \to \wp(A)$*, for some* $g : A \to A$*, and* $g_\sharp : D \to D$. *Then,* $\langle g^\diamond, g_\sharp \rangle$ *is sound iff* $\mathbb{T}_{\text{PCGC}}(\langle g^\diamond, g_\sharp \rangle)$ *is sound w.r.t.* $\mathbb{T}_{\text{PCGC}}(\mathcal{D})$.
(3) *The transforms* $\mathbb{T}_{\text{PPGC}}$ *and* $\mathbb{T}_{\text{PCGC}}$ *are one the inverse of the other.*

Since $f_\sharp$ is defined to be sound when $\eta^\vee \circ f^\diamond \circ \mu \sqsubseteq f_\sharp$ holds, it is then natural to define $f_\sharp$ to be optimal when $\eta^\vee \circ f^\diamond \circ \mu = f_\sharp$, backward complete when $\eta^\vee \circ f^\diamond = f_\sharp \circ \eta^\vee$ and forward complete when $f^\diamond \circ \mu = \mu \circ f_\sharp$. In particular, these definitions allow us to apply the abstraction refinement operators introduced in [14] for minimally refining the abstract domain B in order to obtain a backward/forward complete abstract function and the technique introduced in [12,13] for simplifying abstract domains while retaining the optimality of abstract operations.

6.2 An Example of PCGC

Consider the following infinite complete lattice $\langle B, \leq \rangle$.



B is intended to be an abstract domain which includes both constant and sign information of an integer variable. Indeed B can be defined as the well-known reduced product [5] of the standard constant propagation domain [19] and of the abstraction Sign in Example 4.1. For example, for a while program such as:

$$x := 2; y := 2; \textbf{ while } x < 9 \textbf{ do } x := x * y;$$

a standard analysis with this abstract domain B allows us to derive the loop invariant $\{x > 0,\ y = 2\}$.

It turns out that the abstraction B can be constructively defined. This definition of B relies on $\eta : \mathbb{Z} \to B$ and $\mu : B \to \wp(\mathbb{Z})$ which are essentially defined as identity functions. It should be clear that B is a purely partitioning domain, while it is not a fully partitioning domain, and therefore B cannot be equivalently defined by a constructive Galois connection. In fact, $\mathcal{C} = \langle \eta, \mathbb{Z}, B, \mu \rangle$ is not a CGC, because $1 \in \mu({>}\,0)$ while $1 = \eta(1) \not\preceq {>}\,0$, so that (CGC-Corr) does not hold. Instead, $\mathcal{C}$ turns out to be a PCGC.

Consider the concrete binary integer multiplication $\otimes : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. By following Theorem 6.6 (1), we define a corresponding abstract multiplication $\otimes_\sharp : B \times B \to B$ as follows:

$$\otimes_\sharp(b_1, b_2) \triangleq \eta^\vee(\otimes^\diamond(\mu(b_1), \mu(b_2)))$$

This means that $\otimes_\sharp$ is defined as best correct approximation of the powerset lifting $\otimes^\diamond : \wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ w.r.t. the PPGC $\langle \eta^\vee, \wp(\mathbb{Z})_\subseteq, B_\leq, \mu \rangle = \mathbb{T}_{\text{PPGC}}(\mathcal{C})$, i.e., $\otimes_\sharp = \eta^\vee \circ \otimes^\diamond \circ \mu$. For instance, we have that $\otimes_\sharp(2, {<}\,0) = {<}\,0$ and $\otimes_\sharp(-2, {\leq}\,0) = {\geq}\,0$. Then, since $\langle \otimes^\diamond, \otimes_\sharp \rangle$ is sound, by construction, for $\mathbb{T}_{\text{PPGC}}(\mathcal{C})$, we have that $\langle \otimes, \otimes_\sharp \rangle$ is sound for $\mathcal{C}$. Furthermore, as expected, it turns out that $\otimes_\sharp$ is backward complete for $\mathcal{C}$, meaning that for any $X, Y \in \wp(\mathbb{Z})$, $\vee_B\{x \otimes y \mid x \in X, y \in Y\} = \otimes_\sharp(\vee_B X, \vee_B Y)$. For instance, we have that:

$$\vee_B\, (\otimes^\diamond(\{2,4\}, \{-1,0\})) = \vee_B\{0, -2, -4\} = \ {\leq}\,0\ = \\ \otimes_\sharp\, ({>}\,0,\ {\leq}\,0) = \otimes_\sharp(\vee_B\{2,4\}, \vee_B\{-1,0\}).$$

7 Conclusion

This paper showed that constructive Galois connections, proposed by Darais and Van Horn [10] as a way to define domains to be used in a mechanized and calculational approach to abstract interpretation, are mathematically isomorphic to an already known class of Galois connections which formalize partitions of an unordered set as an abstract domain. Building on that, we defined a novel class of constructive abstract domains, called purely constructive Galois connections. We showed that this class of abstract domains permits to represent a set partition in a flexible way while preserving a constructive approach to Galois connections.

References

1. S. Blazy, V. Laporte, A. Maroneze, D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Proc. of the 20th International Static Analysis Symposium (SAS'13)*, Springer LNCS 7935, pp. 324-344, 2013.

2. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, eds, *Calculational System Design*, NATO ASI Series F. IOS Press, Amsterdam, 1999.
3. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM POPL*, pp. 238-252, 1977.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM POPL*, pp. 269-282, 1979.
6. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Computation*, 2(4):511-547, 1992.
7. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages) (Invited Paper). In *Proc. of the IEEE Int. Conf. on Computer Languages (ICCL'94)*, pp. 95-112. IEEE Computer Society Press, 1994.
8. P. Cousot and R. Cousot. Abstract interpretation of algebraic polynomial systems. In *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, LNCS 1349, pp. 138-154, 1997.
9. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM POPL*, pp. 84-97, 1978.
10. D. Darais and D. Van Horn. Constructive Galois connections: taming the Galois connection framework for mechanized metatheory. In *Proceedings of the 21st ACM Intern. Conf. on Functional Programming (ICFP'16)*. ACM, pp. 311-324, 2016.
11. R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comp. Program.*, 32:177-210, 1998.
12. R. Giacobazzi and F. Ranzato. Example-guided abstraction simplification. In *Proc. 37th Intern. Colloq. on Automata, Languages and Programming (ICALP'10)*, LNCS 6199, pp. 211-222, Springer, 2010.
13. R. Giacobazzi and F. Ranzato. Correctness kernels of abstract interpretations. *Information and Computation*, 237:187-203, 2014.
14. R. Giacobazzi, F. Ranzato and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361-416, 2000.
15. J.H. Jourdan. *Verasco: a Formally Verified C Static Analyzer*. PhD thesis, Université Paris Diderot (Paris 7), France, 2016.
16. J.H. Jourdan, V. Laporte, S. Blazy, X. Leroy, D. Pichardie. A formally-verified C static analyzer. In *Proc. 42nd ACM POPL*, pp. 247-259, 2015.
17. J. Midtgaard and T. Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *Proc. 15th Intern. Static Analysis Symposium (SAS'08)*, LNCS 5079, pp. 347-362, Springer, 2008.
18. D. Monniaux. *Réalisation Mécanisée d'Interpréteurs Abstraits*. Rapport de DEA, Université Paris VII, France, 1998. In French.
19. F. Nielson, H.R. Nielson, C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
20. D. Pichardie. *Interprétation Abstraite en Logique Intuitionniste: Extraction d'Analyseurs Java Certifiés*. PhD thesis, Université de Rennes, France, 2005. In French.
21. F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In *Proc. 13th European Symposium on Programming (ESOP'04)*, LNCS. 2986, pp. 18–32, Springer, 2004.
22. F. Ranzato and F. Tapparo. An abstract interpretation-based refinement algorithm for strong preservation. In *Proc. 11th Intern. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, LNCS 3440, pp. 140–156, Springer, 2005.

23. F. Ranzato and F. Tapparo. Generalized strong preservation by abstract interpretation. *J. Logic and Computation*, 17(1):157-197, 2007.
24. P.F. Silva and J.N. Oliveira. 'Galculator': Functional prototype of a Galois-connection based proof assistant. In *Proc. 10th International ACM Conference on Principles and Practice of Declarative Programming (PPDP'08)*, pp. 44-55, ACM, 2008.

Revisiting MITL to Fix Decision Procedures

Nima Roohi[1] and Mahesh Viswanathan[2]

[1] Department of Computer and Information Science
University of Pennsylvania**
roohi2@cis.upenn.edu
[2] Department of Computer Science
University of Illinois Urbana-Champaign
vmahesh@illinois.edu

Abstract. *Metric Interval Temporal Logic (MITL)* is a well studied real-time, temporal logic that has decidable satisfiability and model checking problems. The decision procedures for MITL rely on the automata theoretic approach, where logic formulas are translated into equivalent timed automata. Since timed automata are not closed under complementation, decision procedures for MITL first convert a formula into negated normal form before translating to a timed automaton. We show that, unfortunately, these 20-year-old procedures are incorrect, because they rely on an incorrect semantics of the $\mathcal{R}$ operator. We present the right semantics of $\mathcal{R}$ and give new, correct decision procedures for MITL. We show that both satisfiability and model checking for MITL are EXPSPACE-complete, as was previously claimed. We also identify a fragment of MITL that we call $\mathtt{MITL}_{\mathrm{WI}}$ that is richer than $\mathtt{MITL}_{0,\infty}$, for which we show that both satisfiability and model checking are PSPACE-complete. Many of our results have been formally proved in PVS.

1 Introduction

Specifications for real time systems often impose quantitative timing constraints between events that are temporally ordered. Classical temporal logics such as Linear Temporal Logic (LTL) [12] are therefore not adequate. Among the many *real-time* extensions of LTL, the most well studied is *Metric Temporal Logic (MTL)* [7]. The temporal modalities in this logic, like $\mathcal{U}_{\mathtt{I}}$, are constrained by a time interval I which requires that the second argument of the $\mathcal{U}$ operator be satisfied in the interval I. For example, the MTL formula in Equation 1 specifies that, at all times, every request should be followed by a response within 5 units of time, or in case there is no response during that time, an error should be raised within the next 0.1 units of time.

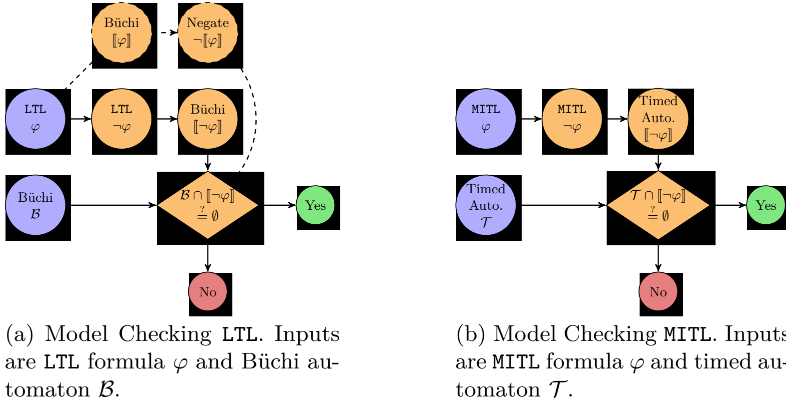
$$\Box\ \mathtt{req} \rightarrow \left(\Diamond_{[0,5)}\mathtt{resp} \vee \left(\Box_{[0,5)}\neg\mathtt{resp} \ \wedge\ \Diamond_{(5,5.1]}\mathtt{error}\right)\right) \quad (1)$$

Classical decision problems for any logic are satisfiability and model checking. For MTL, these problems are highly undecidable; both problems are Σ_1^1-complete [2]. Undecidability in these cases arises because of specifications that

** Part of this work was carried out while the first author was at the University of Illinois, Urbana-Champaign.

© Springer International Publishing AG 2018
I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 474–494, 2018.
https://doi.org/10.1007/978-3-319-73721-8_22

require events to happen exactly at certain time points, which can be described in the logic by having temporal operators decorated by singleton intervals (i.e., intervals containing exactly one point). If one considers the sublogic of MTL, called *Metric Interval Temporal Logic (MITL)*, which prohibits the use of singleton intervals, then both these problems are claimed to be EXPSPACE-complete [2].



(a) Model Checking LTL. Inputs are LTL formula φ and Büchi automaton $\mathcal{B}$.

(b) Model Checking MITL. Inputs are MITL formula φ and timed automaton $\mathcal{T}$.

Fig. 1. Model Checking Steps for LTL and MITL Formulas.

The decision procedures for satisfiability and model checking of MITL, follow the automata theoretic approach. In the automata theoretic approach to satisfiability or model checking, logical specifications are translated into automata such that the language recognized by the automaton is exactly the set of models of the specification. In case of LTL, this involves translating formulas to Büchi automata, and the model checking procedure is shown in Figure 1a. For MITL, formulas are translated into timed automata. Model checking timed automata against MITL specifications is schematically shown in Figure 1b. The specification φ is negated, a timed automaton $[\![\neg\varphi]\!]$ for $\neg\varphi$ is constructed, and then one checks that the system represented as a timed automaton $\mathcal{T}$ has an empty intersection with $[\![\neg\varphi]\!]$. Since timed automata are not closed under complementation [1, 9], MITL decision procedures crucially rely on transforming specifications φ (for the satisfaction problem) and $\neg\varphi$ (for the model checking problem) into negated normal form, *i.e.*, one where all the negations have been pushed all the way inside to only apply to propositions. Using negated normal forms requires considering formulas with the full set of boolean operators (both $\wedge$ and $\vee$) and temporal operators (both $\mathcal{U}$ and its dual $\mathcal{R}$).

Unfortunately, the well known decision procedures for MITL [2] are *incorrect*. This is because we show that the semantics used for the $\mathcal{R}$ operator, which is lifted from the semantics of $\mathcal{R}$ in LTL, is not the dual of $\mathcal{U}$ (see Example 5). Therefore, the timed automata constructed for the negated normal form of a formula in MITL, is not logically equivalent to the original formula. We give a new, correct semantics for $\mathcal{R}$ (Definition 6 in Section 3). Defining the semantics

for $\mathcal{R}$ in MTL is non-trivial because of the subtle interplay of open and closed intervals. Our definition uses 3 quantified variables (unlike 2, which is used in the semantics of $\mathcal{U}$ in both LTL and MTL, and $\mathcal{R}$ in LTL and the incorrect definition for MTL). We show that under fairly general syntactic conditions, one cannot use 2 quantified variables to correctly define the semantics of $\mathcal{R}$ in MTL.

We present a new translation of MITL formulas into timed automata that uses our new semantics (Section 4). We show, using our new construction, that the complexity of the satisfiability and model checking problems remains EXPSPACE-complete as was previous claimed [2]. $\mathtt{MITL}_{0,\infty}$ is a fragment of MITL that has PSPACE decision procedures for satisfiability and model checking. Our last result (Section 5) shows that $\mathtt{MITL}_{0,\infty}$ can be generalized. We introduce a new, richer fragment of MITL that we call $\mathtt{MITL}_{\mathrm{WI}}$, for which we prove that satisfiability and model checking are both PSPACE-complete.

Proofs for results about MITL are subtle due to the presence of a continuous time domain and topological aspects like open and closed sets. This is evidenced by the fact that the errors we have exposed in this paper, have remained undiscovered for over 20 years, despite many researchers working on problems related to MITL. Therefore, to gain greater confidence in the correctness of our claims, we have formally proved many of our results in PVS [11]. The PVS proof objects can be downloaded from http://uofi.box.com/v/PVSProofsOfMITL.

Related Work. The complexity of satisfiability and model checking of MITL formulas was first considered in [2]. We show that the decision procedures are unfortunately flawed because of the use of an incorrect semantics for $\mathcal{R}$. A different translation of MITL to timed automata is presented in [8]. However, their setting is restricted in that all intervals are closed, and all signals are continuous from the right. Note that Example 5 and Theorem 12 in our paper, both use signals that are not continuous from right. Therefore, their algorithm does not fix the problem in [2]. Papers [4,5] present decision procedures for an event-based semantics for MITL which associates a time with every event. State-based semantics, considered here and in [2,8], is very different. For example, in a signal where p is only true in the interval $[0, c]$, there is no time that can be ascribed to the event when p first becomes false. A recent survey of results concerning MTL and its fragments can be found in [10]. Finally, robust model checking of coFlat-MTL formulas with respect to sensor and environmental noise, is considered in [3].

2 Preliminaries

Sets and Functions. The set of *natural*, *positive natural*, *real*, *positive real*, and *non-negative real* numbers are respectively denoted by $\mathbb{N}$, $\mathbb{N}_{+}$, $\mathbb{R}$, $\mathbb{R}_{+}$, and $\mathbb{R}_{\geq 0}$. For a set A, power set of A is denoted by 2^A, Cartesian product of sets A and B is denoted by $A \times B$. Cardinality of A is denoted by $|A|$. The set of functions from A to B is denoted by $A \to B$. For a set A, we denote the fact that a is an element of A by the notation $a : A$. If A is a subset of $\mathbb{R}$ then for any $\epsilon : \mathbb{R}_{\geq 0}$, we

define $\mathtt{B}^{\epsilon}_{\infty}(A) := \{x : \mathbb{R} \mid \exists a : A \bullet |x - a| \leq \epsilon\}$ to be the ϵ-ball around A. Finally, for any two numbers $a, b : \mathbb{R}$, we define $a \dot{-} b$ to be $\max\{a - b, 0\}$.

Intervals. Every *interval* of real numbers is specified by a constraint of the form $a \lhd_1 x \lhd_2 b$, where $a : \mathbb{R} \cup \{-\infty\}$, $b : \mathbb{R} \cup \{\infty\}$, and $\lhd_1, \lhd_2 : \{<, \leq\}$. We use the usual notation $[a, b]$, (a, b), $(a, b]$, and $[a, b)$ to denote *closed*, *open*, *left-open*, or *right-open* intervals. The set of *intervals* and *non-negative intervals* over $\mathbb{R}$, are denoted by $\mathcal{I}$ and $\mathcal{I}_{\geq 0}$, respectively. For any interval $\mathtt{I}$, we use $\underline{\mathtt{I}}$ and $\overline{\mathtt{I}}$ to respectively denote *infimum* and *supremum* of $\mathtt{I}$; if $\mathtt{I}$ is empty, $\underline{\mathtt{I}} = \infty$ and $\overline{\mathtt{I}} = -\infty$. *Width* of an interval, denoted by $\|\mathtt{I}\|$, is defined to be $\overline{\mathtt{I}} - \underline{\mathtt{I}}$. Thus the width of the empty interval is $-\infty$. Finally, an interval with only one element is called a *singleton*; the width of such an interval (by the above definition) is 0.

For any interval $\mathtt{I} : \mathcal{I}$, we use $\langle \mathtt{I}]_? := \overline{\mathtt{I}} \notin \mathbb{R} \vee \overline{\mathtt{I}} \in \mathtt{I}$ to check if $\mathtt{I}$ is closed from right. Similarly, we use ${}_?[\mathtt{I}\rangle$ and ${}_?(\mathtt{I}\rangle$ to check if $\mathtt{I}$ is closed/open from left. We use $(\!|\mathtt{I}|\!) := \mathtt{I} \setminus \{\underline{\mathtt{I}}, \overline{\mathtt{I}}\}$ to denote the interval which is achieved after removing infimum and supremum of $\mathtt{I}$ from it. We also use the following intervals: $[\mathtt{I}\rangle := \mathtt{I} \cup \{\underline{\mathtt{I}}\}$; $[\mathtt{I}|\!) := (\mathtt{I} \cup \{\underline{\mathtt{I}}\}) \setminus \{\overline{\mathtt{I}}\}$; and $(\!|\mathtt{I}] := (\mathtt{I} \cup \{\overline{\mathtt{I}}\}) \setminus \{\underline{\mathtt{I}}\}$.

Signal. Throughout this paper, $\mathtt{AP}$ is a non-empty set of *atomic propositions* [3]. *Signal* is any function of type $\mathbb{R}_{\geq 0} \to 2^{\mathtt{AP}}$. Therefore, each signal is function that defines the set of atomic propositions that are true at each instant of time. For a signal f and time point $r : \mathbb{R}_{\geq 0}$, we define $f^r : \mathbb{R}_{\geq 0} \to 2^{\mathtt{AP}} : t \mapsto f(r + t)$ to be another signal that shifts f by r.

2.1 Metric Temporal Logic

In this section, we first define the syntax of metric temporal logic (MTL) and its subclasses metric interval temporal logic (MITL), and metric temporal logic with restricted intervals ($\mathtt{MITL}_{0,\infty}$). We then define the current semantics of these logics from the literature and call this the *old* semantics. Finally, we define the transformation to a negated normal form (nnf) and the finite variability condition (fvar) that are used in decision procedures for these logics.

Definition 1 (Syntax of MTL, MITL, and $\mathtt{MITL}_{0,\infty}$). *Syntax of a MTL formula is defined using the following BNF grammar, where by p and $\mathtt{I}$, we mean an element of $\mathtt{AP}$ and $\mathcal{I}_{\geq 0}$.*

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \mathcal{U}_{\mathtt{I}} \varphi \mid \varphi \mathcal{R}_{\mathtt{I}} \varphi$$

We assume $\neg$ has the highest precedence. Syntax of a MITL formula is the same, except that singleton intervals cannot be used. Finally, $\mathtt{MITL}_{0,\infty}$ is the sublogic of MITL where for every interval $\mathtt{I}$ appearing in a formula, $\underline{\mathtt{I}} = 0$ or $\overline{\mathtt{I}} = \infty$ hold.

Definition 2, gives the semantics of MTL that was introduced in [2] and is commonly used in the literature. Since MITL and $\mathtt{MITL}_{0,\infty}$ are sublogics of MTL, their semantics follow from the semantics of MTL. Later in Example 5, we show

[3] In Section 3.1 and Example 11, we require $|\mathtt{AP}| > 1$.

that this is not the right semantics because $\mathcal{U}$ and $\mathcal{R}$ are *not* duals of each other. In Definition 6, we introduce a new semantics that fixes this problem. We distinguish the two semantics by putting words OLD and NEW, in gray, below the satisfaction relation $\models$ (Definition 2 uses $\models_{\texttt{OLD}}$ and Definition 6 uses $\models_{\texttt{NEW}}$).

Definition 2 (Semantics of MTL). *Let $f : \mathbb{R}_{\geq 0} \to 2^{\texttt{AP}}$ be an arbitrary signal. For a MTL formula φ, satisfaction relation $f \models_{\texttt{OLD}} \varphi$ is defined using the following inductive rules:*

$$
\begin{array}{lll}
f \models_{\texttt{OLD}} \top & & \textit{is true} \\
f \models_{\texttt{OLD}} \bot & & \textit{is false} \\
f \models_{\texttt{OLD}} p & \textit{iff} & p \in f(0) \\
f \models_{\texttt{OLD}} \neg\varphi & \textit{iff} & \neg(f \models_{\texttt{OLD}} \varphi) \\
f \models_{\texttt{OLD}} \varphi_1 \vee \varphi_2 & \textit{iff} & (f \models_{\texttt{OLD}} \varphi_1) \vee (f \models_{\texttt{OLD}} \varphi_2) \\
f \models_{\texttt{OLD}} \varphi_1 \wedge \varphi_2 & \textit{iff} & (f \models_{\texttt{OLD}} \varphi_1) \wedge (f \models_{\texttt{OLD}} \varphi_2) \\
f \models_{\texttt{OLD}} \varphi_1 \mathcal{U}_{\texttt{I}} \varphi_2 & \textit{iff} & \exists t_1 : \texttt{I} \bullet (f^{t_1} \models_{\texttt{OLD}} \varphi_2) \wedge \forall t_2 : (0, t_1) \bullet f^{t_2} \models_{\texttt{OLD}} \varphi_1 \\
f \models_{\texttt{OLD}} \varphi_1 \mathcal{R}_{\texttt{I}} \varphi_2 & \textit{iff} & \forall t_1 : \texttt{I} \bullet f^{t_1} \models_{\texttt{OLD}} \varphi_2 \quad \vee \\
& & \exists t_1 : \mathbb{R}_+ \bullet (f^{t_1} \models_{\texttt{OLD}} \varphi_1) \wedge \forall t_2 : [0, t_1] \cap \texttt{I} \bullet f^{t_2} \models_{\texttt{OLD}} \varphi_2
\end{array}
$$

Finally, $f \not\models_{\texttt{OLD}} \varphi$ is defined to be $\neg(f \models_{\texttt{OLD}} \varphi)$.

The decision procedures for satisfiability and model checking of MITL introduced in [2], rely on translating the formulas into timed automata. Since timed languages are not closed under complementation [1], complementation cannot be handled as a first-class operation. Instead, one constructs an equivalent formula, where the negations are pushed all the way inside to only apply to propositions. We present this definition of the negation normal form (Definition 3) of a MTL formula next. The implicit assumption is that a formula is semantically equivalent to its negation normal form for certain special signals that are said to be *finitely variable*. We will define finite variability after presenting the definition of negation normal form.

Definition 3 (Negated Normal Form). *For any MTL formula φ, its negated normal form, denoted by $\texttt{nnf}(\varphi)$, is a formula that is obtained by pushing all the negations inside operators. It is formally defined using the following inductive rules (p : AP is an atomic formula, and φ_1 and φ_2 are arbitrary MTL formulas):*

$$
\begin{array}{llll}
\texttt{nnf}(\top) := \top & \texttt{nnf}(\neg\top) := \bot & \texttt{nnf}(\ p) := \ \ p & \\
\texttt{nnf}(\bot) := \bot & \texttt{nnf}(\neg\bot) := \top & \texttt{nnf}(\neg p) := \neg p & \texttt{nnf}(\neg\neg\varphi) := \texttt{nnf}(\varphi)
\end{array}
$$

$$
\begin{array}{llll}
\texttt{nnf}(\varphi_1 \vee \varphi_2) & := \texttt{nnf}(\varphi_1) \vee \texttt{nnf}(\varphi_2) & \texttt{nnf}(\varphi_1 \mathcal{U}_{\texttt{I}} \varphi_2) & := \texttt{nnf}(\varphi_1) \mathcal{U}_{\texttt{I}} \texttt{nnf}(\varphi_2) \\
\texttt{nnf}(\varphi_1 \wedge \varphi_2) & := \texttt{nnf}(\varphi_1) \wedge \texttt{nnf}(\varphi_2) & \texttt{nnf}(\varphi_1 \mathcal{R}_{\texttt{I}} \varphi_2) & := \texttt{nnf}(\varphi_1) \mathcal{R}_{\texttt{I}} \texttt{nnf}(\varphi_2) \\
\texttt{nnf}(\neg(\varphi_1 \vee \varphi_2)) & := \texttt{nnf}(\neg\varphi_1) \wedge \texttt{nnf}(\neg\varphi_2) & \texttt{nnf}(\neg(\varphi_1 \mathcal{U}_{\texttt{I}} \varphi_2)) & := \texttt{nnf}(\neg\varphi_1) \mathcal{R}_{\texttt{I}} \texttt{nnf}(\neg\varphi_2) \\
\texttt{nnf}(\neg(\varphi_1 \wedge \varphi_2)) & := \texttt{nnf}(\neg\varphi_1) \vee \texttt{nnf}(\neg\varphi_2) & \texttt{nnf}(\neg(\varphi_1 \mathcal{R}_{\texttt{I}} \varphi_2)) & := \texttt{nnf}(\neg\varphi_1) \mathcal{U}_{\texttt{I}} \texttt{nnf}(\neg\varphi_2)
\end{array}
$$

The semantics of the modal operators $\mathcal{U}$ and $\mathcal{R}$ are defined using quantifiers, and both of them are $\exists\forall$ formulas. However, $\mathcal{U}$ and $\mathcal{R}$ are supposed to be duals of each other (see Definition 3) eventhough they are defined using formulas with the same quantifier alternation. Thus, $\mathcal{U}$ and $\mathcal{R}$ work as duals only for special signals that are *finitely variable* [2,6,10]. Intuitively, it means during any finite amount of time, number of times a signal changes its value is finite. Definition 4 formalizes this condition.

Definition 4 (Finite Variability). *For an implicitly known satisfaction relation* $\models$*, a signal* $f : \mathbb{R}_{\geq 0} \to 2^{\mathtt{AP}}$ *is said to be* finitely variable from right with respect to a MTL formula φ*, denoted by* $\mathtt{fvar_R}(f, \varphi)$*, iff*

$$\forall r : \mathbb{R}_{\geq 0} \bullet \left(\forall \epsilon : \mathbb{R}_{+} \bullet \exists t : (r, r+\epsilon) \bullet f^t \models \varphi\right) \Rightarrow \left(\exists \epsilon : \mathbb{R}_{+} \bullet \forall t : (r, r+\epsilon) \bullet f^t \models \varphi\right)$$

f *is said to be* finitely variable from left with respect to a MTL formula φ*, denoted by* $\mathtt{fvar_L}(f, \varphi)$*, iff*

$$\forall r : \mathbb{R}_{+} \bullet \left(\forall \epsilon : (0, r) \bullet \exists t : (r-\epsilon, r) \bullet f^t \models \varphi\right) \Rightarrow \left(\exists \epsilon : (0, r) \bullet \forall t : (r-\epsilon, r) \bullet f^t \models \varphi\right)$$

f *is said to be* finitely variable with respect to a MTL formula φ*, denoted by* $\mathtt{fvar}(f, \varphi)$*, iff* $\mathtt{fvar_L}(f, \varphi) \wedge \mathtt{fvar_R}(f, \varphi)$*.* f *is said to be* finitely variable (from left/right) *iff for any MTL formula* φ*,* f *is finitely variable (from left/right) with respect to* φ*. Whenever we use finite variability, precise definition of* $\models$ *will be clear from the context.*

Finite variability as defined here (Definition 4), is formulated differently than the definition given in [6, 10]. However, the two definitions are equivalent, and we prefer the presentation given here because it makes the quantifier alternation in the definition explicit.

Definition 4 suggests that to establish finite variability of a signal, we need to consider all possible MTL formulas. However, it is known that a signal is finitely variable iff it is finitely variable over all atomic formulas; we will prove that this observation also holds for the new semantics for $\mathcal{R}$ that we define in the next section (Lemma 8).

Every finitely variable signal can be specified using (finite or countably infinite) sequence of intervals paired with subsets of atomic propositions that are true during that interval. For example, $([0,1], \{p\}), ((1,4), \{q\}), ([4,\infty), \{p,q\})$ specifies a signal that is $\{p\}$ during $[0,1]$, $\{q\}$ during $(1,4)$, and $\{p,q\}$ at all other times. All our examples use this representation for (finitely variable) signals.

Equivalence for formulas in MTL will only be considered with respect to finitely variable signals. That is, two MTL formulas φ_1 and φ_2 are said to be *equivalent*, iff for any finitely variable signal f we have $(f \models \varphi_1) \Leftrightarrow (f \models \varphi_2)$; here $\models$ can either be taken to be the relation defined in Definition 2 or the one we will define in the next section (Definition 6).

3 Defining the Semantics of Release

The semantics of release as defined in Definition 2 does not ensure that $\mathcal{R}$ and $\mathcal{U}$ are duals. Example 5 describes a finite variable signal f such that $f \not\models_{\mathtt{OLD}} p\mathcal{U}_{\mathtt{I}}q$ (for propositions p, q and interval $\mathtt{I}$) and $f \not\models_{\mathtt{OLD}} \neg p\mathcal{R}_{\mathtt{I}}\neg q$. Thus, the transformation to negation normal form described in Definition 3 does not preserve the semantics, making decision procedures for satisfiability and model checking outlined in [2]

incorrect. In this section, we identify the correct semantics of the release operator so that the transformation to negation normal form described in Definition 3 is semantically correct. Our semantics for $\mathcal{R}$ is more complicated than the one in Definition 2, in that uses 3 quantified variables. We conclude this section by establishing that this increase in expression complexity is necessary — it is impossible to define the semantics of $\mathcal{R}$ using a $\exists\forall$ formula that uses only two quantified variables.

Example 5. Let $c : (\!|\mathtt{I}|\!)$ be an arbitrary point, and define f to be the signal $([0,c],\{p\}),((c,\infty),\{q\})$. Clearly, $f \not\models_{\text{OLD}} p\mathcal{U}_{\mathtt{I}}q$ and hence $f \models_{\text{OLD}} \neg(p\mathcal{U}_{\mathtt{I}}q)$. On the other hand, $\neg q$ is not true throughout $\mathtt{I}$ and whenever $\neg p$ is true, $\neg q$ is false. Therefore, $f \not\models_{\text{OLD}} \neg p\mathcal{R}_{\mathtt{I}}\neg q$. Thus, Definition 3 does not preserve the semantics, making decision procedures for satisfiability and model checking [2] of MITL that first convert a formula into negation normal form, incorrect.

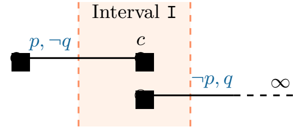


Fig. 2. Signal f

Since the semantics of release is incorrect (from the perspective of ensuring that $\mathcal{U}$ and $\mathcal{R}$ are duals), we define a new semantics for the release operator. Denseness of the time domain, along with subtleties introduced due to open and closed endpoints of intervals, make proofs about MTL challenging to get right. Therefore, to have greater confidence in our results, we have proved most of our results in Prototype Verification Systems (PVS) [11]. We explicitly mark all lemmas and theorems that were proved in PVS [4]. Space limitations prevent these formal proofs to be part of this paper. However they can be downloaded from http://uofi.box.com/v/PVSProofsOfMITL.

Definition 6 (New Semantics for MTL). *Let $f : \mathbb{R}_{\geq 0} \to 2^{\mathsf{AP}}$ be an arbitrary signal. For a MTL formula φ, satisfaction relation $f \models_{\text{NEW}} \varphi$ is defined using the following inductive rules:*

$$
\begin{array}{lll}
f \models_{\text{NEW}} \top & & \textit{is true} \\
f \models_{\text{NEW}} \bot & & \textit{is false} \\
f \models_{\text{NEW}} p & \textit{iff} & p \in f(0) \\
f \models_{\text{NEW}} \neg\varphi & \textit{iff} & \neg(f \models_{\text{NEW}} \varphi) \\
f \models_{\text{NEW}} \varphi_1 \vee \varphi_2 & \textit{iff} & (f \models_{\text{NEW}} \varphi_1) \vee (f \models_{\text{NEW}} \varphi_2) \\
f \models_{\text{NEW}} \varphi_1 \wedge \varphi_2 & \textit{iff} & (f \models_{\text{NEW}} \varphi_1) \wedge (f \models_{\text{NEW}} \varphi_2) \\
f \models_{\text{NEW}} \varphi_1\mathcal{U}_{\mathtt{I}}\varphi_2 & \textit{iff} & \exists t_1 : \mathtt{I} \bullet (f^{t_1} \models_{\text{NEW}} \varphi_2) \wedge \forall t_2 : (0,t_1) \bullet f^{t_2} \models_{\text{NEW}} \varphi_1 \\
f \models_{\text{NEW}} \varphi_1\mathcal{R}_{\mathtt{I}}\varphi_2 & \textit{iff} & \forall t_1 : \mathtt{I} \bullet f^{t_1} \models_{\text{NEW}} \varphi_2 \quad \vee \\
& & \exists t_1 : \mathbb{R}_{+} \bullet (f^{t_1} \models_{\text{NEW}} \varphi_1) \wedge \forall t_2 : [0,t_1] \cap \mathtt{I} \bullet f^{t_2} \models_{\text{NEW}} \varphi_2 \quad \vee \\
& & \exists t_1 : [\mathtt{I}\rangle, t_2 : \mathtt{I} \cap (t_1,\infty) \bullet \forall t_3 : \mathtt{I} \bullet (t_3 \leq t_1 \to f^{t_3} \models_{\text{NEW}} \varphi_2) \\
& & \qquad \wedge (t_1 < t_3 \leq t_2 \to f^{t_3} \models_{\text{NEW}} \varphi_1)
\end{array}
$$

$f \not\models_{\text{NEW}} \varphi$ is defined to be $\neg(f \models_{\text{NEW}} \varphi)$.

Example 7. Consider the signal f from Example 5 that does not satisfy $p\mathcal{U}_{\mathtt{I}}q$. Observe that $f \models_{\text{NEW}} \neg p\mathcal{R}_{\mathtt{I}}\neg q$ by meeting the third condition for satisfying release

[4] Each such result is annotated by ⟨lemma-name⟩@⟨theory-name⟩. Theory name thry can be found in a file named thry.pvs.

operators under the new semantics as follows. Take $t_1 = c$, and $t_2 = c + \epsilon$ such that $[c, c+\epsilon] \subseteq \texttt{I}$. Now, for any $t_3 \leq t_1$, $f^{t_3} \models_{\text{NEW}} \neg q$, and for any $t_1 < t_3 \leq t_2$, $f^{t_3} \models_{\text{NEW}} \neg p$.

We will show that the new semantics (Definition 6) ensures that the transformation to negation normal form (Definition 3) preserves the semantics when considering finite variable signals. Before presenting this result (Theorem 9), we recall that a signal is finitely variable iff the truth of *every* formula in the logic changes only finitely many times within any bounded time. This is difficult to establish. Instead, in [2], it was shown that proving the finite variablity of a signal with respect to atomic propositions, guarantees its finite variability with respect to all formulas. We show that such an observation is also true for the new semantics we have defined.

Lemma 8 PVS[5] **(Finite Variability).** Using the semantics in Definition 6, for any signal f, the following conditions hold:

$$\Big(\forall p : \texttt{AP} \bullet \texttt{fvar}_{\texttt{L}}(f, p)\Big) \Leftrightarrow \Big(\forall \varphi : \texttt{MTL} \bullet \texttt{fvar}_{\texttt{L}}(f, \varphi)\Big)$$
$$\Big(\forall p : \texttt{AP} \bullet \texttt{fvar}_{\texttt{R}}(f, p)\Big) \Leftrightarrow \Big(\forall \varphi : \texttt{MTL} \bullet \texttt{fvar}_{\texttt{R}}(f, \varphi)\Big)$$

We now present the main result about the correctness of the new semantics.

Theorem 9 PVS[6] **(Duality).** If a signal f is finitely variable from right then for any MTL formula φ, $f \models_{\text{NEW}} \varphi$ iff $f \models_{\text{NEW}} \texttt{nnf}(\varphi)$.

We conclude this section by introducing a new (defined) temporal operator that we will use. For any MTL formula φ, let $\bigcirc\varphi$ be defined as $\varphi\mathcal{R}_{(0,\infty)}\varphi$. Intuitively, $f \models_{\text{NEW}} \bigcirc\varphi$ iff φ becomes true and stays true for some positive amount of time. Proposition 10 formalizes this observation. Note that instead if ∞ in definition of $\bigcirc\varphi$, one can use any other positive number and obtain an equivalent formula [7]. In writing formulas, we assume $\bigcirc$ has higher precedence than $\mathcal{U}$ and $\mathcal{R}$ operators but lower precedence than $\vee$ and $\wedge$ operators.

Proposition 10 PVS[7] **(Operator $\bigcirc$).** Let $\models$ be the satisfaction relation given in Definition 2 or Definition 6. For any signal f we have $f \models \bigcirc\varphi$ iff $\exists \epsilon : \mathbb{R}_+ \bullet \forall t : (0, \epsilon) \bullet f^t \models \varphi$.

The correctness of our semantics (Theorem 9) was only established for signals that were finitely variable from the right. Unfortunately, our next example shows that this assumption cannot be relaxed.

Example 11. Let φ be the following formula.

$$(\bigcirc q) \wedge \neg\big(p\mathcal{U}_{(0,1)}q\big) \wedge \neg\big(\neg p\mathcal{U}_{(0,1)}q\big)$$

φ is satisfied by a signal that is finitely variable from the left as follows. Consider f to be such that q is true at all times, and p is true at times $t = \frac{1}{2^n}$ for $n \in \mathbb{N}$ and

[5] `atom_finitevar_eqv_fml_finitevar_left@mtl` and `atom_finitevar_eqv_fml_finitevar_right@mtl`.

[6] `sat_eqv_nnfsat@mtl`.

[7] `next_def_1@mtl`, `next_def_3@mtl`.

false at all other times. First, observe that f is not finitely variable from right (it is, however, finitely variable from left). Also, $f \models \varphi$, no matter whether $\models$ is given by Definition 2 or by Definition 6. Putting φ into its NNF we obtain the following formula which is not satisfiable (using either Definition 2 or Definition 6).

$$(\bigcirc q) \wedge \left(\neg p\mathcal{R}_{(0,1)}\neg q\right) \wedge \left(p\mathcal{R}_{(0,1)}\neg q\right)$$

3.1 Necessity of Using Three Variables

The new semantics of the release operator, given in Definition 6, is defined by quantifying over 3 time points. A natural question to ask is whether this is necessary. Is there a "simpler" definition of the semantics of the release operation? In this section, we show that this is in some sense impossible. We show that no first order definition of the semantics of release that quantifies over only two time points can be correct.

Let us fix the formula $\varphi = \neg(p\mathcal{U}_{\mathtt{I}}q)$, where p and q are proposition. The goal is to show that $\neg\varphi$ cannot be expressed by a "simple" $\exists\forall$-formula. Let us first define what we mean by "simple" formulas. Let $\mathcal{L}_{p,q,\mathtt{I}}$ be the collection of first order formulas of the form

$$\bigwedge_{i:\{1,\ldots,n\}} \bigvee_{j:\{1,\ldots,i_n\}} \exists t_1 : \mathbb{R} \bullet \forall t_2 : \mathbb{R} \bullet \phi_{i,j}(f, t_1, t_2) \tag{2}$$

Here $n : \mathbb{N}$ and $i_n : \mathbb{N}$, and formula $\phi_{i,j}$ is given by the BNF grammar

$$\phi ::= \neg\phi \mid \phi \vee \phi \mid \alpha_1 t_1 + \alpha_2 t_2 \bowtie \beta \mid f^{t_u} \models \psi$$

where $\alpha_1, \alpha_2, \beta : \mathbb{R}$ are arbitrary constants, $\bowtie$: $\{<, \leq, >, \geq\}$ is an arbitrary relation symbol, $u : \{1, 2\}$, and $\psi : \{p, q, \neg p, \neg q\}$. We assume $\models$ here is either $\models_{\text{OLD}}$ or $\models_{\text{NEW}}$; it doesn't make a difference because ψ is propositional. We call constraints of the form $\alpha_1 t_1 + \alpha_2 t_2 \bowtie \beta$ *domain* constraints and constraints of the form $f^{t_u} \models \psi$ *signal* constraints.

Before presenting the main theorem of this section, we examine the restrictions imposed on formulas in $\mathcal{L}_{p,q,\mathtt{I}}$. The requirements that $\phi \in \mathcal{L}_{p,q,\mathtt{I}}$ be in conjuctive normal form, or that there be no $\vee$ or $\wedge$ operations between quantifiers, or that ψ in the BNF only be $\{p, q, \neg p, \neg q\}$ do not restrict the expressive power. Any formula not satifying these conditions can be transformed into one that does. The main restrictions are that all domain constraints are linear and that f in the signal constraints only be shifted by t_1 or t_2 and not by an arithmetic combination of them.

Theorem 12. *There is no formula in $\mathcal{L}_{p,q,\mathtt{I}}$ that is logically equivalent to $\neg(p\mathcal{U}_{\mathtt{I}}q)$ over finite variable signals. In fact, for any $\phi \in \mathcal{L}_{p,q,\mathtt{I}}$, there are signals f_1 and f_2 in which the truth of any atomic proposition changes at most 2 times such that $f_1 \models \neg(p\mathcal{U}_{\mathtt{I}}q)$, $f_2 \models (p\mathcal{U}_{\mathtt{I}}q)$ but either both f_1, f_2 satisfy ϕ or neither does.*

The rest of the section is devoted to proving Theorem 12. Suppose (for contradiction)

$$\phi = \bigwedge_{i:\{1,\ldots,n\}} \bigvee_{j:\{1,\ldots,i_n\}} \exists t_1 : \mathbb{R} \bullet \forall t_2 : \mathbb{R} \bullet \phi_{i,j}(f, t_1, t_2)$$

is logically equivalent to $\neg(p\mathcal{U}_{\mathtt{I}}q)$. We begin by observing that ϕ can be assumed to be in a special canonical form. We then identify two parameters r and δ that are used in the construction of signals that demonstrate the inequivalence of ϕ and $\neg(p\mathcal{U}_{\mathtt{I}}q)$. Finally, we use these parameters to construct the signals and prove the inequivalence.

Cannonical Form of ϕ. We can assume without loss of generality, that ϕ has the following special form.

1. Negations are pushed all the way inside, and are only applied to p or q. This is always possible since $\{<, \leq, >, \geq\}$ is closed under negation and $\neg(f^t \models \psi)$ is, by de nition, equivalent to $f^t \models \neg\psi$. Note that after this step, $\phi_{i,j}$ may contain $\wedge$ operator.
2. Each $\phi_{i,j}$ is a conjunction of clauses that we denote as $\phi_{i,j,k}$.
3. Every clause in $\phi_{i,j}$, has at most one signal constraint of the form $f^{t_1} \models \psi_1$ and one signal constraint of the form $f^{t_2} \models \psi_2$ where ψ_1 and ψ_2 are boolean combinations of p and q.
4. For an arbitrary clause $\phi_{i,j,k}$ in $\phi_{i,j}$, let S and P be, respectively, the set of signal and *negated* domain constraints in $\phi_{i,j,k}$. $\phi_{i,j,k}$ is equivalent to $(\bigwedge_{\theta:P}\theta) \rightarrow (\bigvee_{s:S} s)$. The left hand side of this implication de nes a 2-dimensional convex polyhedron using variables t_1 and t_2.
   For the rest of the proof, *wlog.*, we assume every clause in every $\phi_{i,j}$ is of the form $P \rightarrow S$, where P is a polyhedron over t_1 and t_2, and S is a disjunction of 0, 1, or 2 signal constraints. For any polyhedron P, we de ne $[\![P]\!] := \{(t_1, t_2) \mid P(t_1, t_2)\}$ to be the set of points in P. Also, $\mathtt{cl}([\![P]\!])$ is de ned to be the closure of $[\![P]\!]$. Finally, let $\mathcal{P}$ be the set of all polyhedra used in ϕ.

Figure 3 shows a geometrical interpretation of the polyhedral representation of clauses in $\phi_{i,j}$. Let $\phi_{i,j,k}$ be a clause that is speci ed by $P \rightarrow S$. An arbitrary horizontal line L, may or may not have intersection with P. Either way, L *witnesses* $\exists t_1 \bullet \forall t_2 \bullet \phi_{i,j,k}$ i for all points in this possibly empty intersection, S is satis ed. Every $\phi_{i,j}$ is a set of constraints of the form $P \rightarrow S$. Therefore, L *witnesses* $\exists t_1 \bullet \forall t_2 \bullet \phi_{i,j}$ i L witnesses all clauses in $\phi_{i,j}$. Furthermore, $\exists t_1 \bullet \forall t_2 \bullet \phi_{i,j}$ is true i there is a horizontal line L that witnesses it.

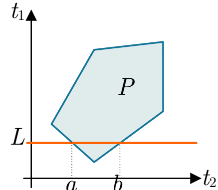
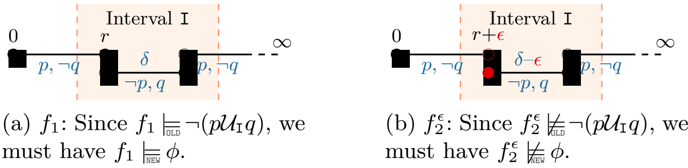


Fig. 3.

Identifying Parameters δ and r. For any $P : \mathcal{P}$, de ne V_P to be the set of vertices of P, and L_P to be the set of points on *vertical* edges of P (that is, segments of a line of the form $t_2 = c$ for some $c : \mathbb{R}$). Let $C_1 := \bigcup_{P:\mathcal{P}}(V_P \cup L_P)$. De ne $C_2 := \{t_2 : \mathbb{R} \mid \exists t_1 : \mathbb{R} \bullet (t_1, t_2) \in C_1\}$ be the projection of points in C_1 on t_2. Take $C_3 := C_2$ if $\overline{\mathtt{I}} = \infty$, and $C_3 := C_2 \cup \{\overline{\mathtt{I}}\}$, otherwise. Observe that C_3 is always a nite set. Therefore, for some $\epsilon : \mathbb{R}_+$, $\mathtt{I} \setminus \mathtt{B}^{\epsilon}_{\infty}(C_3) \neq \emptyset$. Fix $r : \mathtt{I} \setminus C_3$ such that for some ϵ, $\mathtt{B}^{\epsilon}_{\infty}(r) \subseteq \mathtt{I} \setminus C_3$.

For any $P : \mathcal{P}$ and $c : \mathbb{R}$, $\mathtt{cl}([\![P]\!]) \cap [\![t_1 = c]\!]$ is equal to $\{c\} \times J$, for some (possibly empty) interval J. De ne $\|\mathtt{cl}([\![P]\!]) \cap [\![t_1 = c]\!]\|$ to be $\|J\|$. The main



(a) f_1: Since $f_1 \models_{\mathtt{OLD}} \neg(p\mathcal{U}_{\mathtt{I}}q)$, we must have $f_1 \models_{\mathtt{NEW}} \phi$.

(b) f_2^ϵ: Since $f_2^\epsilon \not\models_{\mathtt{OLD}} \neg(p\mathcal{U}_{\mathtt{I}}q)$, we must have $f_2^\epsilon \not\models_{\mathtt{NEW}} \phi$.

Fig. 5. Signals f_1 and f_2^ϵ (interval widths are not proportional).

property we exploit about our choice of r, is that if $(r, c) \in J$ then $\|J\|$ is either ≤ 0 or "large". This is the content of our next lemma.

Lemma 13. *There is a* $\delta : \mathbb{R}_+$ *such that for any* $P : \mathcal{P}$ *and* $c : \mathbb{R}$*, we have that if* $(c, r) \in \mathtt{cl}([\![P]\!])$ *then* $\|\mathtt{cl}([\![P]\!]) \cap [\![t_1 = c]\!]\| \notin (0, \delta]$.

Proof. For the purpose of contradiction, let us assume that the lemma does not hold. Since $\mathcal{P}$ is a nite set, we therefore have,

$$\exists P : \mathcal{P} \bullet \forall n : \mathbb{N}_+ \bullet \exists c_n : \mathbb{R} \bullet (c_n, r) \in \mathtt{cl}([\![P]\!]) \wedge \|\mathtt{cl}([\![P]\!]) \cap [\![t_1 = c_n]\!]\| \in \left(0, \frac{1}{n}\right)$$

Let P be the polyhedron witnessing the violation of the lemma as in the above equation. If $[\![P]\!]$ is an empty set, point or a line/line segment/half line that is not horizontal then its intersection with $[\![t_1 = c_n]\!]$ is either empty or has width 0, which contradicts the fact that P violates the lemma. Otherwise, if $[\![P]\!]$ is a horizontal line, a horizontal half line, or a horizontal line segment, its intersection with $[\![t_1 = c_n]\!]$ is either empty or has a xed width, which again contradicts P violating the lemma. Therefore, consider the case when P has a non-empty interior. Since P has a nite number of vertices, an in nite subsequence of (c_n, r) converges to a vertex of P. However, this is also a contradiction since our choice of r ensures that (c_n, r) is always ϵ away from any point in C_1.

For the rest of this section, let us x r as above, and take δ to be such that in addition to Lemma 13, it satis es $\overline{\mathtt{I}} - r > \delta$. Figure 4 shows a geometric interpretation for the parameters r, δ we have identi ed. For any clause $P \to S$ in ϕ and for any horizontal line L de ned by $t_1 = c$ (for any $c : \mathbb{R}$), if $(c, r) \in [\![P]\!]$ then we have 1. either $a = b = r$, or 2. if S contains a constraint of the form $f^{t_2} \models \psi$ then then S is checked for all values of t_2 in an inveral of size $> \delta$ around r.

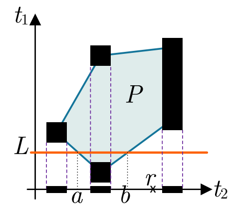


Fig. 4.

Constructing Signal f_1. Figure 5a shows the signal f_1. f_1 is the signal $([0, r], \{p, \neg q\}), ((r, r + \delta], \{\neg p, q\}), ((r + \delta, \infty), \{p, \neg q\})$. It is easy to see that $f_1 \models \neg(p\mathcal{U}_{\mathtt{I}}q)$ (where $\models$ is either $\models_{\mathtt{OLD}}$ or $\models_{\mathtt{NEW}}$); the reason is similar to Example 5. Therefore, if ϕ is equivalent to $\neg(p\mathcal{U}_{\mathtt{I}}q)$, then f_1 also satis es ϕ.

Constructing signal f_2^ϵ. Since f_1 satisfies ϕ, there are $c_1, c_2, \ldots c_n : \mathbb{R}$ and $j_1, \ldots, j_n : \mathbb{N}$ such that for any $i : \{1, \ldots, n\}$, line $L_i \coloneqq (t_1 = c_i)$ witnesses the satisfaction of $\exists t_1 \bullet \forall t_2 \bullet \ \phi_{i,j_i}$. Consider a clause $\phi_{i,j_i,k}$ of ϕ_{i,j_i} of the form $P_{i,j_i,k} \to S_{i,j_i,k}$. We know that line L_i witnesses the satisfaction of this clause. Define interval $\mathtt{J}_{i,j_i,k}$ to be the interval given by $[\![P_{i,j_i,k}]\!] \cap [\![L_i]\!] = \{c_i\} \times \mathtt{J}_{i,j_i,k}$. Choose $\epsilon_{i,j_i,k} : \mathbb{R}_+$ to be such that either $(r, r+\epsilon_{i,j_i,k}) \subseteq \mathtt{J}_{i,j_i,k}$ or $(r, r+\epsilon_{i,j_i,k}) \cap \mathtt{J}_{i,j_i,k} = \emptyset$. Such a choice of $\epsilon_{i,j_i,k}$ always exists no matter what $\mathtt{J}_{i,j_i,k}$ is. Fix the parameter ϵ to be

$$\epsilon \coloneqq \frac{1}{2} \min(\{\epsilon_{i,j_i,k} \mid \text{any } i, j_i, k\} \cup \{c_i - r \mid c_i > r\}).$$

Notice that our choice of ϵ ensures that for i, j_i, k, $(r, r+\epsilon)$ is either contained in $\mathtt{J}_{i,j_i,k}$ or is disjoint from it.

Having defined ϵ, we are ready to describe the signal f_2^ϵ which is shown in Figure 5b. f_2^ϵ is given as $([0, r+\epsilon), \{p, \neg q\}), ([r+\epsilon, r+\delta], \{\neg p, q\}), ((r+\delta, \infty), \{p, \neg q\})$. Notice f_1 and f_2^ϵ only differ in the interval $(r, r+\epsilon)$. Further, f_2^ϵ satisfies $p\mathcal{U}_{\mathtt{I}}q$.

Deriving a Contradiction. Let $c_1, c_2, \ldots c_n$, $L_1, L_2, \ldots L_n$, and $j_1, \ldots j_n$, as defined above, be the witness that demonstrates that f_1 satisfies ϕ. We will show that these also witness the fact that f_2^ϵ satisfies ϕ, giving us the desired contradiction. That is, we will show that the lines $L_i \coloneqq (t_1 = c_i)$ witness the fact that f_2^ϵ satisfies $\exists t_1 \bullet \forall t_2 \bullet \ \phi_{i,j_i}$. Consider any clause $P_{i,j_i,k} \to S_{i,j_i,k}$ of ϕ_{i,j_i}.

- Suppose $S_{i,j_i,k}$ is of the form $f^{t_1} \models \psi$, where ψ is a boolean combination of propositions p, q. Observe that by construction $t_1 \notin (r, r+\epsilon)$, and so $f_1(t_1) = f_2^\epsilon(t_1)$. Therefore, since f_1 satisfies $S_{i,j_i,k}$, so does f_2^ϵ.
- Suppose $S_{i,j_i,k}$ is of the form $f^{t_2} \models \psi$. Let $\mathtt{J}_{i,j_i,k}$ be as defined above. By our choice of ϵ, we know that either $(r, r+\epsilon) \cap \mathtt{J}_{i,j_i,k} = \emptyset$ or $(r, r+\epsilon) \subseteq \mathtt{J}_{i,j_i,k}$. In the first case, we have $f_1(t) = f_2^\epsilon(t)$ for all $t \in \mathtt{J}_{i,j_i,k}$. Therefore, f_1 satisfies $\forall t_2 \in \mathtt{J}_{i,j_i,k} \bullet \ S_{i,j_i,k}$ iff f_2^ϵ satisfies the same condition. Now, let us consider the more interesting case when $(r, r+\epsilon) \subseteq \mathtt{J}_{i,j_i,k}$. Observe that in this case $r \in \mathtt{cl}([\![\mathtt{J}_{i,j_i,k}]\!])$, and so Lemma 13 applies, and we can conclude that $\|\mathtt{cl}([\![\mathtt{J}_{i,j_i,k}]\!])\| > \delta$. This means that either there is a $t < r$ such that $t \in \mathtt{J}_{i,j_i,k}$ or there is a $t > r+\delta$ such that $t \in \mathtt{J}_{i,j_i,k}$. Thus, for any $t_2 \in (r, r+\epsilon)$, there is a $t \in \mathtt{J}_{i,j_i,k}$ such that $f_2^\epsilon(t_2) = f_1(t)$. Hence, once again we can conclude that L_i witnesses the satisfaction of $\forall t_2 \bullet S_{i,j_i,k}$ by f_1^ϵ since f_1 does.
- The last case to consider is when $S_{i,j_i,k}$ is of the form $f^{t_1} \models \psi_1 \vee f^{t_2} \models \psi_2$. In this case also we can conclude that f_2^ϵ satisfies this clause using the reasoning in the previous two cases.

4 Satisfiability and Model Checking MITL Formulas

The satisfiability and model checking problems for MITL are as follows. In satisfiability, given an MITL formula φ, one needs to determine if there is a finite variable signal f that satisfies φ. The model checking problem is, given a timed

automaton $\mathcal{T}$ and a MITL formula φ, determine if every finite variable signal produced by $\mathcal{T}$ satisfies φ. Algorithms for both these problems rely on translating the MITL formula φ (or its negation, in the case of model checking) to a timed automaton $[\![\varphi]\!]$ and then either checking emptiness of $[\![\varphi]\!]$ (for satisfiability) or checking the emptiness of the intersection of two timed automata (for model checking). Since timed automata are not closed under complementation, decision procedures rely on translating a formula in NNF. As observed in Example 5, since the semantics of $\mathcal{R}$ is incorrect, the decision procedures for satisfiability and model checking given in [2,6,10] are incorrect. In this section, we describe a translation of MITL to timed automata with respect to the correct semantics given in Definition 6.

The translation given in [2] from MITL in NNF to timed automata, is correct when the semantics of $\mathcal{R}$ is taken to be as given in Definition 2. We will exploit this construction to give a translation with respect to the semantics in Definition 6. More precisely, in Definition 14, we transform an MITL formula φ into $\texttt{old}(\varphi)$ such that for any signal f, we have $(f \models_{\text{NEW}} \varphi) \Leftrightarrow (f \models_{\text{OLD}} \texttt{old}(\varphi))$.

Definition 14. *The transformation* old *is inductively defined as follows. In this definition,* φ'_1 *and* φ'_2 *are* $\texttt{old}(\varphi_1)$ *and* $\texttt{old}(\varphi_2)$*, respectively.*

$$
\begin{array}{lll}
\texttt{old}(\top) := \top & \texttt{old}(p) \quad := p & \texttt{old}(\varphi_1 \vee \varphi_2) := \varphi'_1 \vee \varphi'_2 \\
\texttt{old}(\bot) := \bot & \texttt{old}(\neg\varphi) := \neg\texttt{old}(\varphi) & \texttt{old}(\varphi_2 \wedge \varphi_2) := \varphi'_1 \wedge \varphi'_2
\end{array}
$$

$$
\begin{array}{lll}
\texttt{old}(\varphi_1 \mathcal{U}_{\mathtt{I}} \varphi_2) & := \varphi'_1 \mathcal{U}_{\mathtt{I}} \varphi'_2 & \\
\texttt{old}(\varphi_1 \mathcal{R}_{\mathtt{I}} \varphi_2) & := (\varphi'_1 \mathcal{R}_{\mathtt{I}} \varphi'_2) \vee (\bigcirc\varphi'_1 \mathcal{R}_{\mathtt{I}} \varphi'_2) & \textit{if } \underline{\mathtt{I}} > 0 \\
& \quad\; (\varphi'_1 \mathcal{R}_{\mathtt{I}} \varphi'_2) \vee (\bigcirc\varphi'_1 \mathcal{R}_{\mathtt{I}} \varphi'_2) \vee \bigcirc\varphi'_1 & \textit{if } \underline{\mathtt{I}} = 0 \wedge {?}(\mathtt{I}\rangle \\
& \quad\; (\varphi'_1 \mathcal{R}_{\mathtt{I}} \varphi'_2) \vee (\bigcirc\varphi'_1 \mathcal{R}_{\mathtt{I}} \varphi'_2) \vee (\varphi'_2 \wedge \bigcirc\varphi'_1) & \textit{otherwise}
\end{array}
$$

The transformation old ensures that the semantics of the transformed formula $\texttt{old}(\varphi)$ with respect to $\models_{\text{OLD}}$, is the same as the semantics of φ with respect to $\models_{\text{NEW}}$.

Lemma 15 $_{\text{PVS}}$[8]**.** For any signal f and MTL formula φ, we have $(f \models_{\text{NEW}} \varphi) \Leftrightarrow (f \models_{\text{OLD}} \texttt{old}(\varphi))$.

Proof (idea). Use induction on the structure of φ. Special treatment for $\underline{\mathtt{I}}$ is needed because both Definition 2 and Definition 6 define what is called *strict* semantics, in which value of signal at time 0 is *not* important when $\underline{\mathtt{I}} > 0$. ☐

It is worth emphasizing that Definition 14 and Lemma 15 apply to any MTL formula (not just MITL), and the soundness of the transformation holds for *any* signal (and not just finite variable signals).

Lemma 15 immediately gives us a procedure for transforming a negated normal formula into a timed automaton according to Definition 6. For any MITL formula φ, we transform $\texttt{old}(\texttt{nnf}(\varphi))$ into a timed automaton according to [2]. Note that output of old is in NNF iff its input is [9]. Using Theorem 9 and Lemma 15 we know that the transformation is correct.

[8] sat_and_isat@mtl.

[9] toISatNNF@mtl.

The main problem with this approach and Definition 14, is that $\mathtt{old}(\varphi)$ could be exponentially larger than φ. So we need to address the concern that this might change the complexity of satisfiability and model checking. The complexity of the transformation in [2] for MITL and $\mathtt{MITL}_{0,\infty}$ depends only on the number of *distinct* subformulas in φ, and *not* on the formula size of φ itself [10]. In Proposition 16, we show that the number of subformulas of $\mathtt{old}(\varphi)$ is *linearly* related to the number of subformulas of φ. Thus using the construction in [2] for $\mathtt{old}(\mathtt{nnf}(\varphi))$ does not change the complexity results for satisfiability and model checking [11].

Proposition 16. *For any MTL formula φ, we have $|\mathcal{S}_{\mathtt{old}(\varphi)}| \leq 6|\mathcal{S}_\varphi|$, where for any MTL formula ψ, $\mathcal{S}_\psi$ is the set of subformulas of ψ (including ψ, itself).*

It is worth noting that this proposition also applies to any MTL formula and not just MITL. Using Proposition 16, we can conclude that the complexity of satisfiability and model checking remain unchanged in the new semantics.

Corollary 17. *With respect to the semantics in Definition 6, the satisfiability and model checking problems for $\mathtt{MITL}_{0,\infty}$ and MITL are PSPACE-complete and EXPSPACE-complete, respectively.*

5 MITL with Wide Intervals ($\mathtt{MITL}_{\mathrm{WI}}$)

One important result in [2] is the identification of sublogic $\mathtt{MITL}_{0,\infty}$ of MITL, for which the satisfiability and model checking problems are in PSPACE, as opposed to EXPSPACE for MITL. In this section we prove that this result can be generalized. We identify a more expressive sublogic of MITL for which satisfiability and model checking are in PSPACE.

For a formula φ of MITL, the size of φ is the size of the formula, where the constants appearing in the intervals are represented in binary. Here we do not restrict constants in φ to be natural numbers (as in [2]), but instead allow them to be rational numbers; as is standard, we represent a rational number as a pair of binary strings encoding the numerator and denominator of the fractional representation. Define $\mathtt{MITL}_{\mathrm{WI}}$ to be the collection of MITL formulas φ such that every interval I appearing in φ, has 1. $\underline{\mathtt{I}} = 0$ or 2. $\overline{\mathtt{I}} = \infty$ or 3. $\frac{\overline{\mathtt{I}}}{\overline{\mathtt{I}}-\underline{\mathtt{I}}} \leq n$ when $0 < \underline{\mathtt{I}} < \overline{\mathtt{I}} < \infty$, where n is the size of φ.

Notice that every $\mathtt{MITL}_{0,\infty}$ formula is also a $\mathtt{MITL}_{\mathrm{WI}}$ formula, and there are many $\mathtt{MITL}_{\mathrm{WI}}$ formulas that are not $\mathtt{MITL}_{0,\infty}$ formulas. Thus, $\mathtt{MITL}_{\mathrm{WI}}$ is a richer fragment of MITL. Condition 3 above in the definition of $\mathtt{MITL}_{\mathrm{WI}}$ says that when there is an interval not conforming to the restrictions of $\mathtt{MITL}_{0,\infty}$, and it has a

[10] The complexity depends on the size of the DAG representation of the formula, and not its syntactic representation.

[11] There are multiple initial transformations in [2], and each one of them can make the size of formula exponentially bigger. However, the number of distinct subformulas remains linear to the size of original formula.

large supremum, then the size of the interval must also be large. Thus, intervals in $\texttt{MITL}_{\text{WI}}$ can be thought of as being "wide" (and hence the name). The main result of this section is the following.

Theorem 18. *For any $\texttt{MITL}_{WI}$ formula φ of size n, there is a timed automaton $[\![\varphi]\!]$ satisfying the following properties.*
1. *For any finite variable signal f, f is in the language of $[\![\varphi]\!]$ Iff $f \models_{\text{NEW}} \varphi$.*
2. *$[\![\varphi]\!]$ has at most $2^{\mathcal{O}(n^2)}$ many locations and edges.*
3. *$[\![\varphi]\!]$ has at most $\mathcal{O}(n^2)$ clocks.*
4. *$[\![\varphi]\!]$ has at most $\mathcal{O}(n)$ distinct integer constants, each bounded by $2^{\mathcal{O}(n)}$.*

Furthermore, $[\![\varphi]\!]$ can be constructed in polynomial space from φ.

The proof of this result will be presented over the course of this section, but it is worth noting that Theorem 18 immediately gives a PSPACE algorithm for satisfiability and model checking of $\texttt{MITL}_{\text{WI}}$.

Corollary 19. *Model checking and satisfiability problems for $\texttt{MITL}_{WI}$ is PSPACE-complete.*

Proof. Being in PSPACE is an immediate consequence of Theorem 18, and PSPACE-hardness follows from the PSPACE-hardness of $\texttt{MITL}_{0,\infty}$. ☐

The rest of this section is devoted to proving Theorem 18. We begin by highlighting the crucial features of $\texttt{MITL}_{0,\infty}$ that make it easier to decide than MITL (Section 5.1). In Section 5.2, we sketch the proof of Theorem 18, by drawing on the observations in Section 5.1.

5.1 MITL vs. $\texttt{MITL}_{0,\infty}$

The algorithm (from [2]) for constructing a timed automaton for a MITL formula φ applies a series of syntactic transformations to φ such that the resulting formula 1. is in negated normal form, 2. has at most linearly more distinct subformulas, 3. has the same maximum constant as the original formula, and most importantly, 4. is in the normal form given in Definition 20. These transformations can be carried out in polynomial time, and the construction of the timed automaton assumes that the MITL formula is in the normal form given by Definition 20 below.

Definition 20 (Normal Form [2, Definition 4.1]). *MITL formula φ is said to be in* normal form *iff it is built from propositions and negated propositions using conjunction, disjunction, and temporal subformulas of the following six types:*

1. *$\Diamond_{\mathtt{I}}\varphi'$ with $\wr(\mathtt{I}\rangle$, $\underline{\mathtt{I}} = 0$, and $\overline{\mathtt{I}} \in \mathbb{R}$,*
2. *$\Box_{\mathtt{I}}\varphi'$ with $\wr(\mathtt{I}\rangle$, $\underline{\mathtt{I}} = 0$, and $\overline{\mathtt{I}} \in \mathbb{R}$,*
3. *$\Box\varphi'$.*
4. *$\varphi_1\mathcal{U}_{\mathtt{I}}\varphi_2$ with $\underline{\mathtt{I}} > 0$, and $\overline{\mathtt{I}} \in \mathbb{R}$,*
5. *$\varphi_1\mathcal{R}_{\mathtt{I}}\varphi_2$ with $\underline{\mathtt{I}} > 0$, and $\overline{\mathtt{I}} \in \mathbb{R}$,*
6. *$\varphi_1\mathcal{U}\varphi_2$,*

The main challenge (in terms of complexity) is in handling formulas of Type 4 and Type 5. If the formula you start with is in $\mathtt{MITL}_{0,\infty}$ then it can be seen that the normal form does not have any subformulas of Type 4 and Type 5. Hence, the timed automaton constructed for $\mathtt{MITL}_{0,\infty}$ formulas is "small", which results in PSPACE decision procedures.

To see the difficulty of transforming Type 4 and Type 5 formulas into timed automata, consider $\Box_{(3,4)}(p \rightarrow \Diamond_{(1,2)} q)$. Intuitively, the formula says, during the 4th unit of time, every p should be followed by a q within 1 to 2 units of time. A naïve approach, starts and dedicates a clock after seeing every p during 4th unit of time, and uses that clock to ensure that there will be at least one q, 1 to 2 units of time after the corresponding p was seen. However, this approach does not work, since there is no bound on number of p that one can expect to see during any period of time, which makes number of required clocks infinite.

Instead, the construction in [2] divides $\mathbb{R}_{\geq 0}$ into $[0,1),[1,2),\ldots$ intervals. Two important facts are central to the construction. 1. For any interval $[n, n+1)$ and any Type 4 $\mathcal{U}$ or Type 5 $\mathcal{R}$ formula φ, the subset of times in $[n, n+1)$ for which φ is true is exactly union of two possibly empty intervals. Using this property, for each interval $[n, n+1)$, we first guess those two intervals and then use at most 4 clocks to verify our guess. 2. We can start reusing a clock at most $\overline{\mathtt{I}}$ units of time after we started using it. Therefore, total number of clocks required for checking each Type 4 and Type 5 formula is bounded by $4\overline{\mathtt{I}}$. Since $\overline{\mathtt{I}}$ could be exponentially big, the resulting timed automaton could have exponentially many clocks [12].

5.2 Witness Points and Intervals

Let us define *step size* of an interval $\mathtt{I}$ as follows.

$$\mathtt{sz}(\mathtt{I}) \coloneqq \begin{cases} \overline{\mathtt{I}} - \underline{\mathtt{I}} & \text{if } \overline{\mathtt{I}} - \underline{\mathtt{I}} < \underline{\mathtt{I}} \\ \underline{\mathtt{I}} & \text{otherwise} \end{cases} \tag{3}$$

The crucial observation needed to prove Theorem 18 is that the truth of Type 4 and Type 5 does not change very frequently. We show that for a bounded non-empty interval $\mathtt{I}$ with $\underline{\mathtt{I}} > 0$, using constantly many clocks, the timed automaton can monitor the truth of a formula of Type 4 or Type 5 for intervals $[0, c), [c, 2c), \ldots$, where $c \coloneqq \mathtt{sz}(\mathtt{I})$, instead of intervals $[0,1),[1,2),\ldots$ as in the construction given in [2]. This has two important consequences.

1. If a formula is in $\mathtt{MITL}_{\mathrm{WI}}$, then number of required clocks will be at most linear in the size of formula. For example, verifying $\Box_{[n,2n]}\varphi$ requires constant number of clocks, as opposed to exponentially many clocks in [2].

[12] The construction in [2], keeps track of the subset of clocks that are free (*i.e.* can be reused) in the discrete modes of the timed automaton. This makes the number of locations doubly exponential. However, it is possible to reuse clocks in a queue like fashion and instead of encoding a subset of free clocks in discrete modes, one can just encode the index of the next free clock. This approach exponentially decreases number of required discrete modes. This optimization however does not change the asymptotic complexity.

2. Consider satisfiability of $\varphi := \Box_{[1,2]}\Diamond_{[0.01,0.02]}\varphi'$ formula. The algorithm in [2], first changes φ to an "equivalent" formula $\varphi := \Box_{[100,200]}\Diamond_{[1,2]}\varphi'$, because if observation intervals are $[0,1),[1,2),\ldots$ then all constants in the input formula must be natural numbers. Therefore, timed automaton will have hundreds of clocks. However, we show there is no need for observation intervals to have natural numbers as endpoints. This means that the timed automaton for φ requires at most 8 clocks for each of $\Box_{[1,2]}$ and $\Diamond_{[0.01,0.02]}$ sub-formulas. Note that the algorithm to check emptiness of timed automata will replace all rational numbers by natural numbers by scaling, when constructing the region graph [1]. However, in spite of this, it is worth observing that the complexity of emptiness checking of timed automata has an exponentially worse dependence on the number of clocks, than on constants [1, Lemma 4.5].

Witness Points for $\mathcal{U}$ Operators. For the rest of this section, let us fix an arbitrary signal f. We begin by presenting some technical definitions of "witnesses" that demonstrate when an $\mathcal{U}$-formula is satisfied.

Definition 21 (Witness Sets for $\mathcal{U}$). *For every* MTL *formulas φ_1 and φ_2, and $i : \{1,2,3\}$, we define* $\texttt{witness}^i_{\mathcal{U}}(\varphi_1,\varphi_2)$ *to be a subset of $\mathbb{R}^2_{\geq 0}$ defined by the following predicates over (r,w):*

1. $r \leq w \wedge \left(\forall t : (r,w) \bullet f^t \models_{\text{NEW}} \varphi_1\right) \wedge f^w \models_{\text{NEW}} \varphi_2$
2. $r < w \wedge \exists \epsilon : \mathbb{R}_+ \bullet \left(\forall t : (r,w) \bullet f^t \models_{\text{NEW}} \varphi_1\right) \wedge \left(\forall t : (w \dot{-} \epsilon, w) \bullet f^t \models_{\text{NEW}} \varphi_2\right)$
3. $r < w \wedge \exists \epsilon : \mathbb{R}_+ \bullet \left(\forall t : (r,w+\epsilon) \bullet f^t \models_{\text{NEW}} \varphi_1\right) \wedge \left(\forall t : (w,w+\epsilon) \bullet f^t \models_{\text{NEW}} \varphi_2\right)$

Notice, that if (r,w) is in any of the witness sets given in Definition 21, then it provides proof that certain until formulas are true. This is captured by the definition of proof sets, given next.

Definition 22 (Proof Sets for $\mathcal{U}$). *For every* MTL *formulas φ_1 and φ_2, times $r,w : \mathbb{R}_{\geq 0}$, interval $\mathtt{I} : \mathcal{I}_{\geq 0}$, and $i : \{1,2,3\}$, we define* $\texttt{proofset}^i_{\mathcal{U}}(\varphi_1,\varphi_2,r,w,\mathtt{I})$ *to be a subset of $\mathbb{R}_{\geq 0}$ defined by the following predicates over t:*

1. $\texttt{witness}^1_{\mathcal{U}}(r,w) \wedge r \leq t \wedge w - t \in \mathtt{I}$
2. $\texttt{witness}^2_{\mathcal{U}}(r,w) \wedge r \leq t \wedge w - t \in (\!|\mathtt{I}]$
3. $\texttt{witness}^3_{\mathcal{U}}(r,w) \wedge r \leq t \wedge w - t \in [\mathtt{I}|\!)$

A proof set $\texttt{proofset}^i_{\mathcal{U}}(\varphi_1,\varphi_2,r,w,\mathtt{I})$ establishes the fact that $\varphi_1\mathcal{U}_{\mathtt{I}}\varphi_2$ is true at time r in signal f. This is proved next.

Proposition 23 $_{\text{PVS}}$[13] **(Proof Sets for $\mathcal{U}$).** For any MTL formulas φ_1 and φ_2, times $r,w : \mathbb{R}_{\geq 0}$, interval $\mathtt{I} : \mathcal{I}_{\geq 0}$, $i : \{1,2,3\}$, and $t : \texttt{proofset}^i_{\mathcal{U}}(\varphi_1,\varphi_2,r,w,\mathtt{I})$ we have $f^t \models_{\text{NEW}} \varphi_1\mathcal{U}_{\mathtt{I}}\varphi_2$.

In Proposition 23, the signal f need not be finitely variable. Also, the formulas φ_1,φ_2 could be any MTL formulas. The truth of $\varphi_1\mathcal{U}_{\mathtt{I}}\varphi_2$ within $[0,\texttt{sz}(\mathtt{I}))$ changes

[13] def_until_witness_{1,2,3}_proofset@mtl_witness.

only finitely many times. This crucial observation helps limit the number of clocks needed to monitor the truth of $\mathcal{U}$-subformulas.

Theorem 24 $_{\texttt{PVS}}$[14] **(Finite Variability of $\mathcal{U}$).** For any MTL formulas φ_1 and φ_2, and interval $\mathtt{I} : \{\mathtt{I} : \mathcal{I} \mid \underline{\mathtt{I}}, \overline{\mathtt{I}} \in \mathbb{R}_+\}$, there are two intervals $T_1 : \mathcal{I}_{\geq 0}$ and $T_2 : \mathcal{I}_{\geq 0}$ with the following properties:
- $\forall t_1 : T_1, t_2 : T_2 \bullet t_1 < t_2$, and
- $\forall t : \mathbb{R}_{\geq 0} \bullet (t < \mathtt{sz}(\mathtt{I}) \wedge f^t \models_{\texttt{NEW}} \varphi_1 \mathcal{U}_{\mathtt{I}} \varphi_2) \Leftrightarrow (t \in T_1 \cup T_2)$

It is worth noting that Theorem 24 is not restricted to MITL or to finite variable signals. Since within $[0, \mathtt{sz}(\mathtt{I}))$, the times when $\varphi_1 \mathcal{U}_{\mathtt{I}} \varphi_2$ is true can be partitioned into two intervals, suggests that a timed automaton checking this property can just guess these intervals. But how can such intervals be guessed? Definition 21 provides an answer. These observations are combined in the next theorem, to identify what the timed automaton needs to guess and check for $\mathcal{U}$-formulas.

Theorem 25 $_{\texttt{PVS}}$[15] **(Witness Point for $\mathcal{U}$).** In Theorem 24, if $\mathtt{fvar}(f)$ then T_1 and T_2 have the following properties:
- If $T_1 \neq \emptyset$ then there are $w_1 : \mathbb{R}_+$ and $i : \{1, 2\}$ such that:
  1. if $i = 1$ then $w_1 - \underline{T_1} \in \mathtt{I}$, otherwise, $w_1 - \underline{T_1} \in \langle \mathtt{I}]$
  2. $(\underline{T_1}, w_1) \in \mathtt{witness}^i_{\mathcal{U}}(\varphi_1, \varphi_2)$
  3. $T_1 \subseteq \mathtt{proofset}^i_{\mathcal{U}}(\varphi_1, \varphi_2, \underline{T_1}, w_1)$
- If $T_2 \neq \emptyset$ then there are $w_2 : \mathbb{R}_+$ and $i : \{1, 3\}$ such that:
  1. $w_2 - \underline{T_2} \in \langle \mathtt{I}]$
  2. $(\underline{T_2}, w_2) \in \mathtt{witness}^i_{\mathcal{U}}(\varphi_1, \varphi_2)$
  3. $T_2 \subseteq \mathtt{proofset}^i_{\mathcal{U}}(\varphi_1, \varphi_2, \underline{T_2}, w_2)$

In Theorem 25, the 1$^{\text{st}}$ property bounds possible values of w_i, and hence bounds possible values that should be guessed by timed automaton. The 2$^{\text{nd}}$ property specifies what w_i should satisfy (*i.e.* what timed automaton should verify about the guess), and the 3$^{\text{rd}}$ property states that w_i is enough for proving that $\varphi_1 \mathcal{U}_{\mathtt{I}} \varphi_2$ is satisfied by f at all times in T_i.

Witness Intervals for $\mathcal{R}$ Operators. We now identify how a timed automaton can check $\mathcal{R}$-formulas. We will repeat the steps from the previous section. We will identify witness intervals, and proof sets for $\mathcal{R}$-formulas. As in the case of $\mathcal{U}$, these provide proofs of when a $\mathcal{R}$ formula is true.

Definition 26 (Witness Interval for $\mathcal{R}$). *For every MTL formulas φ_1 and φ_2, and $i : \{1, \ldots, 4\}$, we define $\mathtt{witness}^i_{\mathcal{R}}(\varphi_1, \varphi_2)$ to be a subset of $\mathcal{I}$ defined by the following predicates over* I*:*

1. $\forall t : \mathtt{I} \bullet f^t \models_{\texttt{NEW}} \varphi_2$
2. $\mathtt{I} \neq \emptyset \wedge {}_{¿}[\mathtt{I}\rangle \wedge \forall t : \mathtt{I} \bullet f^t \models_{\texttt{NEW}} \varphi_1$
3. $\mathtt{I} \neq \emptyset \wedge \langle \mathtt{I}]_{?} \wedge \forall t : \mathtt{I} \bullet f^t \models_{\texttt{NEW}} \varphi_2 \wedge f^{\overline{\mathtt{I}}} \models_{\texttt{NEW}} \varphi_1$
4. $[\mathtt{I}\rangle \neq \emptyset \wedge \langle \mathtt{I}]_{?} \wedge \forall t : \mathtt{I} \bullet f^t \models_{\texttt{NEW}} \varphi_2 \wedge \exists \epsilon : \mathbb{R}_+ \bullet \forall t : (\overline{\mathtt{I}}, \overline{\mathtt{I}} + \epsilon) \bullet f^t \models_{\texttt{NEW}} \varphi_1$

[14] `until_witness_interval_2@mtl_witness`.
[15] `until_witness_interval_3@mtl_witness`.

Definition 27 (Proof Sets for $\mathcal{R}$). *For every MTL formulas φ_1 and φ_2, intervals $\mathtt{I}, \mathtt{J} : \mathcal{I}_{\geq 0}$, and $i : \{1, 2, 3, 4\}$, we define $\mathtt{proofset}^i_{\mathcal{U}}(\varphi_1, \varphi_2, \mathtt{I}, \mathtt{J})$ to be a subset of $\mathbb{R}_{\geq 0}$ defined by the following predicates over $t : \mathbb{R}_{\geq 0}$:*

1. $\mathtt{I} \in \mathtt{witness}^1_{\mathcal{R}}(\varphi_1, \varphi_2) \wedge t + \mathtt{J} \subseteq \mathtt{I}$
2. $\mathtt{I} \in \mathtt{witness}^2_{\mathcal{R}}(\varphi_1, \varphi_2) \wedge t + \mathtt{J} \subseteq (\underline{\mathtt{I}}, \infty) \wedge t < \overline{\mathtt{I}} \wedge \underline{\mathtt{J}} > 0$
3. $\mathtt{I} \in \mathtt{witness}^3_{\mathcal{R}}(\varphi_1, \varphi_2) \wedge t + \mathtt{J} \subseteq \mathtt{I} + \mathbb{R}_{\geq 0} \wedge t < \overline{\mathtt{I}}$
4. $\mathtt{I} \in \mathtt{witness}^4_{\mathcal{R}}(\varphi_1, \varphi_2) \wedge t + \mathtt{J} \subseteq \mathtt{I} + \mathbb{R}_{\geq 0} \wedge t \leq \overline{\mathtt{I}}$

Proposition 28 $_{\mathtt{PVS}}$[16] **(Proof Sets for $\mathcal{R}$).** For any MTL formulas φ_1 and φ_2, intervals $\mathtt{I}, \mathtt{J} : \mathcal{I}_{\geq 0}$, $i : \{1, 2, 3, 4\}$, and $t : \mathtt{proofset}^i_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I}, \mathtt{J})$ we have $f^t \models_{\text{NEW}} \varphi_1 \mathcal{R}_{\mathtt{J}} \varphi_2$.

Like $\mathcal{U}$-formulas, a formula $\varphi_1 \mathcal{R}_{\mathtt{I}} \varphi_2$ changes its truth only finitely many times in the interval $[0, \mathtt{sz}(\mathtt{I}))$.

Theorem 29 $_{\mathtt{PVS}}$[17] **(Finite Variability of $\mathcal{R}$).** For any MTL formulas φ_1 and φ_2, and non-empty positive bounded interval $\mathtt{I}$, there are four intervals $T_1, \ldots, T_4$ with the following properties:

– $\forall i, j : \{1, \ldots, 4\}, t_i : T_i, t_j : T_j \bullet i < j \Rightarrow t_i < t_j$, and
– $\forall t : \mathbb{R}_{\geq 0} \bullet (t < \mathtt{sz}(\mathtt{I}) \wedge f^t \models_{\text{NEW}} \varphi_1 \mathcal{R}_{\mathtt{I}} \varphi_2) \Leftrightarrow (t \in \bigcup_{i:1,\ldots,4} T_i)$

Like in Theorem 25, we can combine Theorem 29 and Definition 26 to come up with how a timed automaton can check such $\mathcal{R}$ formulas.

Theorem 30 $_{\mathtt{PVS}}$[17] **(Witness Interval for $\mathcal{R}$).** In Theorem 29, if $\mathtt{fvar}(f)$ then $T_1, \ldots, T_4$ have the following properties:

– If $T_1 \neq \emptyset$ then $\exists \mathtt{I} : \mathcal{I}$ such that:
  1. $\mathtt{I} \subseteq (0, \mathtt{sz}(\mathtt{J}))$
  2. $\mathtt{witness}^2_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I})$
  3. $T_1 \subseteq \mathtt{proofset}^2_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I}, \mathtt{J})$
– If $T_2 \neq \emptyset$ then $\exists \mathtt{I} : \mathcal{I}$ such that:
  1. $\mathtt{I} \subseteq [\mathtt{sz}(\mathtt{J}), \mathtt{sz}(\mathtt{J}) + \overline{\mathtt{J}})$
  2. $\mathtt{witness}^1_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I})$
  3. $T_2 \subseteq \mathtt{proofset}^1_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I}, \mathtt{J})$
– If $T_3 \neq \emptyset$ then $\exists \mathtt{I} : \mathcal{I}$ such that:
  1. $\mathtt{I} \subseteq [\mathtt{sz}(\mathtt{J}), \mathtt{sz}(\mathtt{J}) + \overline{\mathtt{J}})$
  2. $\mathtt{witness}^4_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I})$
  3. $T_3 \subseteq \mathtt{proofset}^4_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I}, \mathtt{J})$
– If $T_4 \neq \emptyset$ then $\exists \mathtt{I} : \mathcal{I}, i : \{2, 3, 4\}$ such that:
  1. $\mathtt{I} \subseteq [\mathtt{sz}(\mathtt{J}), \mathtt{sz}(\mathtt{J}) + \overline{\mathtt{J}})$
  2. $\mathtt{witness}^i_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I})$
  3. $T_4 \subseteq \mathtt{proofset}^i_{\mathcal{R}}(\varphi_1, \varphi_2, \mathtt{I}, \mathtt{J})$

Constructing a timed automaton for $\mathtt{MITL}_{\mathbf{WI}}$. One can use Theorem 25 and Theorem 30 and follow the same ideas outlined in [2] to construct a timed automaton for Type 4 and Type 5 formulas. Let us outline how this works for Type 4 formulas. If the automaton guesses that T_1 is not empty then it must make this guess at time exactly $\underline{T_1}$. At the same time, the automaton takes two more actions: First, it resets a *free* clock x and remembers that this clock is not free anymore. Second, it guesses whether i should be 1 or 2. Suppose, i is chosen

[16] `def_release_witness_{1,2,3,4}_proofset@mtl_witness`.
[17] `release_witness_interval_1@mtl_witness`.

to be 1. As long as x is not free, the automaton makes sure that the input signal satisfies φ_1. Note that this is a different proof obligation and will be considered by an induction on the structure of input formula. At the same time or at some time later, the automaton should guess whether the current time is $\overline{T_1}$. At any point in time, if the automaton does not make that call (*i.e.* decides the current time is not $\overline{T_1}$), it means the automaton wants to prove that the input signal satisfies the $\mathcal{U}$ formula at all points in time between $\underline{T_1}$ and sometime in the future. As soon as the automaton guesses that the current time is $\overline{T_1}$, it resets a new free clock y and marks it as non-free. The automaton then makes sure that when current values of x and y belong to $\mathtt{I}$, the input signal satisfies φ_2 at least once. As soon as φ_2 becomes true during during this period, the proof obligation is over and x and y will both be marked as free clocks (note that φ_1 does not need to be true when φ_2 becomes true). Using Theorem 25, we know what the automaton checks, guarantees $\forall t : T_1 \bullet f^t \models_{\text{NEW}} \varphi_1 \mathcal{U}_{\mathtt{I}} \varphi_2$. However, the automaton has only finitely many clocks and it cannot reuse a clock while it is not free. The significance of Theorem 24 is that it guarantees simultaneously guessing and proving at most $\lceil \frac{\mathtt{I}}{\mathtt{sz(I)}} \rceil + 1$ number of T_1, T_2 intervals is enough. Since clocks x and y will be freed at most $\overline{\mathtt{I}}$ units of time after they became non-free, number of required clocks for each Type 4 formula will be only twice the number of simultaneous proof obligations. The same argument holds for $\mathcal{R}$ operators, except that the automaton has to simultaneously guess and prove at most $\lceil \frac{\mathtt{I}}{\mathtt{sz(I)}} \rceil + 1$ number of T_1, T_2, T_3, T_4 intervals.

6 Conclusion

We proved that the classical decision procedures for satisfiability and model checking of MITL [2] are incorrect. This is because they rely on a semantics for the $\mathcal{R}$ operator which is not the dual of $\mathcal{U}$. We give a new semantics of $\mathcal{R}$ and prove that it behaves like the dual of $\mathcal{U}$ over signals that are finitely variable. Identifying the right semantics for $\mathcal{R}$ is subtle as we show that it is not possible to give a correct semantics using characterization that uses only two quantified variables. Using the new semantics, we give a translation from MITL to timed automata and thereby correcting the decision procedures for MITL. Finally, we also identify a fragment of MITL called $\mathtt{MITL}_{\mathrm{WI}}$, that is more expressive than $\mathtt{MITL}_{0,\infty}$, but nonetheless has decision procedures in PSPACE.

7 Acknowledgement.

We gratefully acknowledge the support of the following grants: Nima Roohi was partially supported by NSF CNS-1505799 and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy and by ONR N000141712012. Mahesh Viswanathan was partially supported by NSF CCF 1422798, NSF CNS 1329991, and AFOSR FA9950-15-1-0059.

References

1. Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The Benefits of Relaxing Punctuality. *J. ACM*, 43(1):116–146, 1996.
3. Patricia Bouyer, Nicolas Markey, and Pierre-Alain Reynier. Robust Analysis of Timed Automata Via Channel Machines. In *FoSSaCS 2008*.
4. Thomas Brihaye, Morgane Estiévenart, and Gilles Geeraerts. On MITL and Alternating Timed Automata. In *FORMATS*, pages 47–61. Springer Berlin Heidelberg, 2013.
5. Thomas Brihaye, Morgane Estiévenart, and Gilles Geeraerts. On MITL and Alternating Timed Automata over Infinite Words. In *FORMATS*, pages 69–84. Springer International Publishing, 2014.
6. Yoram Hirshfeld and Alexander Rabinovich. Logics for Real Time: Decidability and Complexity. *Fundam. Inf.*, 62(1):1–28, 2004.
7. Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
8. Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to Timed Automata. In *FORMATS*, pages 274–289, 2006.
9. Madhavan Mukund. Finite-state Automata on Infinite Inputs, 1996.
10. Joël Ouaknine and James Worrell. Some Recent Results in Metric Temporal Logic. In *FORMATS*, pages 1–13, 2008.
11. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *CADE*, pages 748–752, 1992.
12. Amir Pnueli. The Temporal Logic of Programs. In *SFCS*, pages 46–57. IEEE Computer Society, 1977.

Selfless Interpolation for Infinite-State Model Checking*

Tanja Schindler[1] and Dejan Jovanović[2]

[1] University of Freiburg
[2] SRI International

Abstract. We present a new method for interpolation in satisfiability modulo theories (SMT) that is aimed at applications in model-checking and invariant inference. The new method allows us to control the finite-convergence of interpolant sequences and, at the same time, provides expressive invariant-driven interpolants. It is based on a novel integration of model-driven quantifier elimination and abstract interpretation into existing SMT frameworks for interpolation. We have integrated the new approach into the SALLY model checker and we include experimental evaluation showing its effectiveness.

1 Introduction

Many modern model-checking techniques rely on Craig interpolation [14,30] as a learning oracle to support abstraction refinement and invariant inference. Interpolants themselves are artifacts usually computed from proofs of correctness for a finite unrolling of the system under analysis. While it is possible for a model checker to compute interpolants on its own, in most cases interpolation is provided by the underlying reasoning engine (such as an SMT solver) that is *unaware of the application-specific needs*. The importance of good interpolants is widely acknowledged – a single "magical" interpolant can make a difference between verifying the model instantaneously and verification failure – and it is no surprise that interpolants and their properties have been studied extensively. Some examples of interpolant properties are interpolant (logical) strength [16,38,34], size [18], and beauty [1].

Interpolant properties mentioned above are conceptually appealing but focus on single interpolants in isolation. In this paper we investigate interpolants as used in the IC3/PDR class of model-checking algorithms (e.g. [4,7,23,22]), in the context of analysis of infinite state systems. The IC3/PDR class of algorithms reasons locally, without unrolling the systems, and constructs abstractions and invariant candidates incrementally. The overall algorithm performs not one, but a sequence of reasoning queries that are interleaved and interact with results of interpolation. Therefore, we are ultimately more interested in *properties of the interpolation sequence*[3] rather than a single interpolant. Ideally, the interpolation

* The research presented in this paper has been supported by NSF grant 1528153.
[3] Not to be confused with sequence interpolants [29].

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 495–515, 2018.
https://doi.org/10.1007/978-3-319-73721-8_23

procedure would offer some convergence guarantees of this reasoning sequence. For example, in the PDKIND method [22] (a variant of PDR), interpolation is used to incrementally refine the current candidate invariant by examining induction failures. If this refinement sequence is allowed to continue indefinitely, the PDKIND method will never get to reason past its current invariant candidate and will fail to make progress. Similarly, in IC3/PDR algorithms, a non-converging interpolation sequence will result in verification failure where all reachability frames are refined indefinitely.

Example 1 (Model Checking Divergence). Consider a simple transition system defined with the initial states and a transition relation:

$$I = (x = 0) \wedge (c = 1) \ , \qquad\qquad T \equiv (x' = x + c) \wedge (c' = 2c) \ .$$

The system above satisfies the invariant $(x \geq 0)$. Nevertheless, well-known interpolation-based model checkers such as NUXMV [5][4] and SPACER [23], and our own tool SALLY [22] diverge and fail to prove the property. □

We propose a new interpolation method that is based on the following two guiding principles:

1. The interpolation method should interact well with the underlying model-checking algorithm and offer guarantees of convergence for interpolation sequences.
2. The interpolation method should be aware of the model checking context and be able to accept suggestions from the model checker in order to produce invariant-driven interpolants.

Our new method is developed within the interpolation framework available in all major SMT solvers (e.g. [10]). For the theory of arithmetic, most SMT solvers produce interpolants through application of the Farkas lemma [35]. We first show how this approach can result in divergence, as in the example above, and then propose a solution based on model-driven quantifier elimination that guarantees convergence. The new method is very flexible and allows the model checker to also provide interpolant suggestions. We use this feature to make the resulting interpolants invariant-directed, by integrating abstract interpretation [12] into the interpolation process. To the best of our knowledge, our interpolation method is the first to combine interpolation, quantifier elimination, and abstract interpretation, in a framework readily reusable in any interpolating SMT solver.

We have implemented our new approach into the SALLY model-checker by relying on the MATHSAT5 SMT solver [9] and the APRON domain library [20]. We present experimental data that shows the effectiveness of the approach and illuminates the impact of quantifier elimination and abstract interpretation.

[4] In order to rely only on interpolation, we disable predicate abstraction in NUXMV.

2 Background

2.1 Satisfiability Modulo Theories

We work in the setting of satisfiability modulo theories (SMT) and assume the usual notation of first order logic (see, e.g. [2]). In the following, we use the letters x, y, z to denote variables, and c to denote constants. We consider the quantifier-free theory of linear arithmetic over the rationals ($\mathcal{T}_{\text{LA}}$). We denote with p and q linear terms over $\mathbb{Q}$, i.e., terms of the form $c_n x_n + \cdots + c_1 x_1 + c_0$ over variables $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$ with coefficients $c_0, \ldots, c_n \in \mathbb{Q}$. A theory atom in arithmetic is a linear constraint, i.e., an inequality of the form $(p \diamond 0)$ with $\diamond \in \{\leq, <\}$,[5] and a literal is an atom or its negation. A clause is a disjunction of literals and we denote with $\perp$ the empty clause. As usual, formulas are constructed inductively from atoms and the usual Boolean connectives. We denote with $\text{ATOMS}(F)$ the set of all atoms appearing in a formula F, and with $\text{LITERALS}(F)$ the set of all literals of F, i.e. $\text{LITERALS}(F) = \{a, \neg a \mid a \in \text{ATOMS}(F)\}$. If formula A has all its free variables in $\boldsymbol{x}$, we denote this with $A(\boldsymbol{x})$. A formula $A(\boldsymbol{x})$ is satisfiable if there is an assignment mapping its variables to real values such that A evaluates to **true** in the usual interpretation. A conjunction of literals C is a $\mathcal{T}_{\text{LA}}$-conflict, if C is inconsistent with $\mathcal{T}_{\text{LA}}$. A $\mathcal{T}_{\text{LA}}$-lemma is a clause C such that $\neg C$ is a $\mathcal{T}_{\text{LA}}$-conflict or, in other words, C is a valid statement in arithmetic. To ease notation, we also treat conjunctions and clauses as sets of literals.

Given a formula F that is unsatisfiable in $\mathcal{T}_{\text{LA}}$, a resolution proof of F is a tree $\mathcal{P}$ such that a) the root of $\mathcal{P}$ is the empty clause, b) leafs of $\mathcal{P}$ are either clauses from F, or $\mathcal{T}_{\text{LA}}$-lemmas, and c) each non-leaf node C is an application of Boolean resolution, i.e. $C \equiv (C_1 \vee C_2)$ and has two parents $(C_1 \vee l)$ and $(\neg l \vee C_2)$ as below.

$$\frac{(C_1 \vee l) \qquad (\neg l \vee C_2)}{(C_1 \vee C_2)}$$

Most modern SMT solvers rely on some variant of the DPLL($\mathcal{T}$) framework [32] to check satisfiability of formulas. In this framework, to solve a formula F, a CDCL SAT solver is used to enumerate the truth values of the propositional skeleton of the formula F. As the formula atoms are assigned to **true** or **false** by the SAT solver, a dedicated decision procedure for the theory $\mathcal{T}$ checks the consistency of the literals $A \subseteq \text{LITERALS}(F)$ corresponding to the Boolean assignment. If A is unsatisfiable, the decision procedure returns a $\mathcal{T}$-conflict $C \subseteq A$ (or equivalently a $\mathcal{T}$-lemma $\neg C$) that explains the inconsistency in Boolean terms to the SAT solver. The basic DPLL($\mathcal{T}$) framework can be extended to also provide proofs of unsatisfiability by stitching together the Boolean reasoning and the $\mathcal{T}$-lemmas resulting from the conflicts. In proofs generated by a DPLL($\mathcal{T}$) solver, by construction, the clauses that make up the proof *only contain atoms from the original formula.* [6]

[5] For simplicity we do not consider equality, since they can be eliminated by rewriting $(p = 0)$ with $(p \leq 0) \wedge \neg(p < 0)$.

[6] In general, SMT solvers may introduce new literals to support reasoning in more expressive theories, but for reasoning in $\mathcal{T}_{\text{LA}}$ this is unnecessary.

In the case of linear real arithmetic, we are talking about DPLL($\mathcal{T}_{\text{LA}}$), and the decision procedure most commonly used is based on a variant of the Simplex algorithm [17] engineered specifically for DPLL($\mathcal{T}_{\text{LA}}$). Besides its efficiency, this Simplex algorithm also has a remarkable property that the $\mathcal{T}_{\text{LA}}$-conflicts that it produces are *minimal*.

2.2 Craig Interpolation

Definition 1 (Craig interpolant). *Given two formulas $A(\boldsymbol{x}, \boldsymbol{y})$ and $B(\boldsymbol{y}, \boldsymbol{z})$ such that $A \wedge B$ is unsatisfiable, a* Craig interpolant *is a formula $J(\boldsymbol{y})$ such that $A \Rightarrow J$ and $J \Rightarrow \neg B$. We call the pair (A, B) an* interpolation problem.

The formulation above is the general version of the interpolation problem. In model checking applications, interpolation problems are often specialized so that the formulas in question are of the form $A(\boldsymbol{x}, \boldsymbol{x}')$ and $B(\boldsymbol{x}')$. In such cases, it is easy to see that $J \equiv \neg B$ is a solution to the interpolation problem, and we call J the *trivial interpolant*.

Interpolants for the interpolation problem (A, B) can be computed from a proof of unsatisfiability of the formula $A \wedge B$. The underlying technique of this proof-based interpolation is generally attributed to Pudlák [19,25,33], and was revisited in recent years in the context of model checking and SMT solving (see, e.g. [27,28,10]). Given a proof of unsatisfiability for $A \wedge B$, the interpolant can be computed inductively over the structure of the proof tree. As the proof tree is traversed from the leaves to the root, each clause C in the proof tree is associated a partial interpolant. A *partial interpolant* of a clause C is an interpolant of the formulas

$$(A \wedge (\neg C \downarrow A)) \ , \qquad\qquad (B \wedge (\neg C \downarrow B)) \ .$$

The projection functions $(\cdot \downarrow A)$ and $(\cdot \downarrow B)$ operate on literals and have the following properties. For a literal l, one of $(l \downarrow A)$ and $(l \downarrow B)$ must be l. The projection $(l \downarrow A)$ may be l, if $l \in \textsc{literals}(A)$, or it must be $\top$ otherwise. Dually, $(l \downarrow B)$ may be l, if $l \in \textsc{literals}(B)$, or it must be $\top$ otherwise. The projection function is extended to conjunctions of literals, as expected. Intuitively, the projection functions are used to extract literals that come from A or B, with some flexibility in the ownership of shared literals.[7] For a clause C, we write $\neg C \setminus A$ as a shortcut for the literals in $\neg C$ without $(\neg C \downarrow A)$, and analogously for B. Note that the partial interpolant of the empty clause $\bot$ is an interpolant of the original problem that can be read of as the partial interpolant associated to the root of the proof. The rules for computing partial interpolants depend on the type of the proof node, as follows.

1. For an input clause C from A the partial interpolant is $\neg(\neg C \setminus A)$.
2. For an input clause C from B the partial interpolant is $\neg C \setminus B$.

[7] This flexibility allows interpolants of different logical strength [16].

3. For a $\mathcal{T}$-lemma C, the conjunction $\neg C$ is unsatisfiable, and the partial interpolant is computed by a $\mathcal{T}$-lemma interpolator as the interpolant of $(\neg C \downarrow A)$ and $(\neg C \downarrow B)$.
4. For a resolvent clause C the partial interpolant J is computed as follows

$$\frac{(C_1 \vee l) : J_1 \qquad (\neg l \vee C_2) : J_2}{(C_1 \vee C_2) : J} \text{ where } J = \begin{cases} J_1 \vee J_2, & \text{if } l \downarrow B = \top, \\ J_2 \wedge J_2, & \text{if } l \downarrow A = \top, \\ ite(l, J_2, J_1), & \text{otherwise.} \end{cases}$$

For a general and more detailed exposition on the overall framework we refer the reader to [6]. Given a $\mathcal{T}$-lemma interpolator P, we denote with $\textsc{itp}[\![P]\!]$ the proof-based interpolation procedure that uses P to interpolate the $\mathcal{T}$-lemma nodes of the proof.

In general, the structure of proof-based interpolants is hard to control: an interpolant will be a Boolean combination of parts of A clauses, parts of B clauses, and the interpolants from the $\mathcal{T}$-lemmas. Nevertheless, we can guarantee that no literals that are exclusively in B can sneak into the interpolant, unless introduced by the lemmas.

Lemma 1. *Given an interpolation problem (A, B), the interpolant $J = \textsc{itp}[\![P]\!]$ is a Boolean combination over the atoms of A and atoms from the $\mathcal{T}$-lemma interpolants.*

Proof. We only have to show that no atom from B, that neither appears in A nor is produced by the $\mathcal{T}$-lemma interpolator, can ever sneak into the final interpolant. This is trivial for the partial interpolants for input clauses from A and theory lemmas. The only atoms that can be added to the interpolant in resolution nodes, are atoms for which $l \downarrow A = l$ (and $l \downarrow B = l$) holds, and hence $l \in \textsc{literals}(A)$. We are left with the case of a proof node that is an input clause from B. In this case the partial interpolant is $J \equiv (\neg C \setminus B)$. By definition, for each $l \in J$ we know that $l \downarrow B = \top$. Therefore we must have $l \downarrow A = l$ and $l \in \textsc{literals}(A)$. □

2.3 Arithmetic Interpolation

In the proof-based interpolation framework, for an SMT solver to provide interpolation in the theory of arithmetic, it needs to be able to provide interpolation for each $\mathcal{T}_{\text{LA}}$-theory lemma that it contributes to the proof. For an interpolation problem (A, B), the lemmas of the proof correspond to $\mathcal{T}_{\text{LA}}$-conflicts that were found by the solver during the solving process. In case of arithmetic, and SMT solvers based on Simplex, each $\mathcal{T}_{\text{LA}}$-conflict will be a set of literals C that is inconsistent (and minimal). Each conflict C can be separated into the A part and B part, and the goal is to find an interpolant for the interpolation problem $(C \downarrow A, C \downarrow B)$.

For linear arithmetic, the most common way to obtain the interpolant of a conflict is to rely on the Farkas lemma. A $\mathcal{T}_{\text{LA}}$-theory conflict C is an unsatisfiable

conjunction of inequalities

$$I_i \equiv (\sum_j c_{ij} x_j + c_{i0} \ \diamond_i \ 0) \ ,$$

for $\diamond_i \in \{<, \leq\}$. By Farkas lemma, there exist coefficients $k_i > 0$ that can certify the inconsistency, i.e. such that

$$\sum_i k_i \times I_i = (1 < 0) \ .$$

The lemma interpolant can then be given by summing up the A contributions to the conflict, i.e., the interpolant is

$$J \equiv \sum_{I_i \in C \downarrow A} k_i \times I_i \ .$$

It is not hard to see that J is a valid interpolant. The advantage of the Farkas approach is that the coefficients k_i can easily be read off the state of the Simplex solver when it detects a conflict. We will denote the $\mathcal{T}_{\text{LA}}$-lemma interpolator based on the Farkas lemma as P_{FK}.

Note that an interpolant obtained with P_{FK} is always *a single inequality.* The ability to produce a single inequality can be advantageous, as it allows the interpolant to relate variables that might not be syntactically related in A, by using the B part. On the other hand, as we will see in the next section, the disadvantage of P_{FK} is that it can lead to diverging interpolant sequences.

3 Sequences of Interpolants

Behavior of interpolant sequences was first explored in [22], where the notion of finite-covering interpolation was proposed as an assumption that supports termination and deductive power of the PDKIND method.

Definition 2 (Finite Covering Interpolation). *An interpolation procedure P (or a $\mathcal{T}$-lemma interpolator) is* finite covering *if for a fixed $A(\boldsymbol{x}, \boldsymbol{y})$, it can only produce a finite number of distinct interpolants.*

Example 2. If the interpolation problems are of the form $A(\boldsymbol{x}, \boldsymbol{x}')$ and $B(\boldsymbol{x}')$, the trivial interpolation method can always return $\neg B$ as the interpolant. This kind of interpolation is not useful in general and is not finite covering.

Finite covering is a strong property. Most interpolation procedures are proof-based and, since the space of proofs and lemmas is infinite, they do not ensure finite covering. Nevertheless, for theories that admit quantifier elimination, for any given A, one can construct a single interpolant $J(\boldsymbol{y})$ by eliminating $\boldsymbol{x}$ from $(\exists \boldsymbol{x} \ . \ A(\boldsymbol{x}, \boldsymbol{y}))$ that refutes any B that needs to be interpolated. In principle, for arithmetic theories, a finite-covering interpolation procedure could be devised

by relying on procedures such as MCSat [15] that are based on quantifier elimination. But, since none of the available interpolating SMT solvers are MCSat-based, it would be desirable to have some control over the number of potential invariants in the existing proof-based interpolation framework.

Definition 3 (Interpolation Sequence). *Given a formula $A(\boldsymbol{x}, \boldsymbol{y})$ and two sequences of formulas $(J_k(\boldsymbol{y}))$ and $(B_k(\boldsymbol{y}, \boldsymbol{z}))$, we call (J_k) an* interpolation sequence *for A and (B_k) if for all k it holds that*

1. *B_k is consistent with $\bigwedge_{i<k} J_i$;*
2. *B_k is inconsistent with A;*
3. *J_k is the interpolant between A and B_k.*

Definition 4 (Finite Convergence). *We say that an interpolation procedure has a* finite convergence property *if it does not allow infinite interpolation sequences.*

To put the definitions above in perspective, in a typical model checking application, the formula A will correspond to some abstraction of reachable states (including the transition relation), formulas B_k will correspond to potentially bad states, and the interpolants J_k will be learned facts that refute the potentially bad states. The finite convergence property then guarantees that no matter how we choose the potentially bad states, the interpolation procedure will eventually refute all of them. Finite convergence differs from finite covering in that it is semantic and more directly addresses the undesirable interpolant behavior.

Example 3 (Finite Convergence). Consider the interpolation procedure $\text{ITP}[\![P_{\text{FK}}]\!]$, i.e. the standard SMT interpolation for $\mathcal{T}_{\text{LA}}$ based on Farkas derivation. Let $A(x, y_1, y_2)$ be the constraints

$$I_1 \equiv (y_1 - x < 0) \ , \qquad I_2 \equiv (x < 0) \ , \qquad I_3 \equiv (y_2 - x < 0) \ .$$

Now, consider the sequence of formulas (B_k), where $B_k(y_1, y_2) \equiv (y_1 + ky_2 > 0)$. Interpolating from A against an individual B_k using the Farkas approach will always result in an interpolant J_k that is a single inequality constructed as a combination of formulas from A, i.e., we will obtain

$$J_k \equiv 1 \times I_1 + (k+1) \times I_2 + k \times I_3 \equiv (y_1 + ky_2 < 0) \ .$$

Since interpolating from A over the sequence (B_k) results in an infinite sequence of distinct interpolants, the Farkas approach to interpolation does not have the finite-covering property, i.e., neither P_{FK} nor $\text{ITP}[\![P_{\text{FK}}]\!]$ guarantee finite convergence.

On the other hand, we can rely on Fourier-Motzkin quantifier elimination to simply eliminate x from A and obtain the conjunction $J \equiv (y_1 < 0) \wedge (y_2 < 0)$ that is a suitable interpolant for all B_k simultaneously (it is derivable from A and singlehandedly refutes all B_k). Note that, as mentioned before, the Farkas-based interpolants relate variables y_1 and y_2 in the interpolants. On the other hand, the interpolants based on Fourier-Motzkin do not. □

An interesting property of finite convergence is that we can interleave an interpolation procedure P_1 with finite convergence with an arbitrary interpolation procedure P_2 and still obtain finite convergence, as long as the interleaving is fair to the procedure P_1.[8]

The following lemma shows that we do not need to devise an entirely new interpolation procedure to ensure finite convergence. Instead, we only need to devise a finite-covering $\mathcal{T}$-lemma interpolator that can then be used in the standard proof-based interpolation framework.

Lemma 2. *If a $\mathcal{T}$-lemma interpolator P is finite covering, then the proof-based interpolation procedure* $\text{ITP}[\![P]\!]$ *has the finite convergence property.*

Proof. Assume that $\text{ITP}[\![P]\!]$ does not have the finite convergence property. This means that there is an infinite interpolation sequence, i.e. there is a formula A, and two sequences of formulas J_k and B_k as in Definition 3. In this sequence, the interpolants J_k must be distinct functions because for each k

- J_k is inconsistent with B_k; but
- J_i is consistent with B_k, for $i < k$.

On the other hand, P can only produce a finite number of lemma interpolants and, by Lemma 1, $\text{ITP}[\![P]\!]$ (as a proof-based procedure) can only produce Boolean combinations of clauses from A and lemma interpolants. Therefore, the overall procedure $\text{ITP}[\![P]\!]$ will only be able to produce a finite number of distinct interpolants (seen as functions), proving the case by contradiction. □

4 Interpolation with Conflict Resolution

In this section we present a $\mathcal{T}_{\text{LA}}$-lemma interpolator P_{CR} that replaces the traditional interpolator based on the Farkas lemma P_{FK}. Throughout this section we therefore assume a global interpolation problem, i.e. formulas $A(\boldsymbol{x}, \boldsymbol{y})$ and $B(\boldsymbol{y}, \boldsymbol{z})$, with $A \wedge B$ unsatisfiable. In addition, we assume a global ordering on variables so that $\boldsymbol{z} \prec \boldsymbol{y} \prec \boldsymbol{x}$. Our goal is to devise the $\mathcal{T}_{\text{LA}}$-lemma interpolator P_{CR} that is finite covering. In order to achieve this we will rely on a model-driven variant of Fourier-Motzkin (FM) quantifier elimination.

4.1 Fourier-Motzkin Elimination

Given an inconsistent set of inequalities F, a FM proof of F has the same structure as a Boolean resolution proof would, but with clauses replaced with inequalities, and the resolution rule replaced with the FM elimination rule. Given two inequalities sharing a variable x of opposite signs, the FM rule deduces a new inequality with this variable eliminated.[9]

[8] In a way, an interpolation procedure that has the finite convergence property is analogous to the widening operator in abstract interpretation.

[9] The presented rule is over strict inequalities only, other cases are as expected.

$$\text{FM } x \; \frac{I_l \equiv (p - x < 0) : \qquad I_u \equiv (x - q < 0)}{R \equiv (p - q < 0)}$$

We first explain the general idea behind the new lemma interpolation procedure. Each $\mathcal{T}_{\text{LA}}$-lemma interpolation problem consists of two sets of inequalities $C_A(\boldsymbol{x}, \boldsymbol{y})$ and $C_B(\boldsymbol{y}, \boldsymbol{z})$, with $C_A \wedge C_B$ unsatisfiable. Therefore, there exists a FM elimination proof of inconsistency that is ordered according to $\prec$. In other words, in the FM proof the $\boldsymbol{x}$ variables are eliminated first, followed by the $\boldsymbol{y}$ variables, and finally the $\boldsymbol{z}$ variables. The order of elimination ensures, for example, that if any inequality I in the proof contains an $\boldsymbol{x}$ variable, I must have been derived from C_A. Let J be the set of inequalities in the proof that do not contain any $\boldsymbol{x}$ variables but were either a) derived from two inequalities that contain $\boldsymbol{x}$ variables; or b) appear in C_A directly. By construction, then $C_A \Rightarrow J$ and J is in $\boldsymbol{y}$ variables only. In addition, the inequalities in J constitute a cut of the proof tree that is enough to refute C_B. In other words, J is an interpolant between C_A and C_B. This selection of inequalities from the proof can be done locally at each resolution node, and we denote the procedure that returns the relevant inequalities as $\textsc{select}(R, I_l, I_u)$.

Example 4. Let's revisit the interpolation problem of Example 3, i.e., let

$$C_A \equiv (y_1 - x < 0) \wedge (x < 0) \wedge (y_2 - x < 0) \;, \qquad C_{B_k} \equiv (y_1 + ky_2 > 0) \;.$$

Below is a Fourier-Motzkin proof of unsatisfiability of $C_A \wedge C_{B_k}$, with the variables ordered as $y_2 \prec y_1 \prec x$. We mark inequalities derived only from C_A with red bold font, and all other inequalities with blue.

$$y_2 \; \dfrac{y_1 \; \dfrac{x \; \dfrac{\mathbf{y_1 - x < 0} \qquad \mathbf{x < 0}}{\mathbf{y_1 < 0}} \qquad -y_1 - ky_2 < 0}{-ky_2 < 0} \qquad \dfrac{x \; \dfrac{\mathbf{y_2 - x < 0} \qquad \mathbf{x < 0}}{\downarrow}}{\mathbf{y_2 < 0}}}{0 < 0}$$

As discussed above, we can examine the proof and get that

$$\textsc{select}((y_1 < 0), (y_1 - x < 0), (x < 0)) = \{(y_1 < 0)\} \;,$$
$$\textsc{select}((y_2 < 0), (y_2 - x < 0), (x < 0)) = \{(y_2 < 0)\} \;.$$

Therefore the set of inequalities $J = \{\ (y_1 < 0), (y_2 < 0)\ \}$ is an interpolant for C_A and C_{B_k} for *any* k. □

4.2 Conflict Resolution

Although we could use FM elimination to derive the $\mathcal{T}_{\text{LA}}$-lemma interpolants, as above, this would likely not be efficient. FM elimination is a quantifier elimination procedure and can become very inefficient even with small numbers of variables. Instead, we will adopt a model-driven variant of FM elimination called *conflict resolution* (CR). The conflict resolution algorithm was originally introduced in [24] for solving systems of linear inequalities. CR is an instance of a

recent class of model-based decision procedures, such as Generalized DPLL [26] and MCSat [15], but is simpler as it targets conjunctions of constraints only. The algorithm is related to FM elimination in the same way the CDCL algorithm is related to Boolean resolution: instead of trying to prove the problem unsatisfiable by saturating the FM rule, conflict resolution attempts to build a model and only applies the FM rule when the model-building fails. This principled way of deriving new inequalities makes it possible to produce a proof while only deducing inequalities that are relevant for unsatisfiability.

We use a variation of the original algorithm [24] adapted to the context of $\mathcal{T}_{\text{LA}}$-lemma interpolation. In this context, we are given two sets of inequalities $C_A(\boldsymbol{x}, \boldsymbol{y})$ and $C_B(\boldsymbol{y}, \boldsymbol{z})$ that together are known to be unsatisfiable. The algorithm will construct a proof that $C_A \wedge C_B$ is unsatisfiable and, as a side-effect, collect the set of inequalities J that will form the interpolant of C_A and C_B. Before we describe the algorithm itself, we go trough some of its ingredients.

We order all the variables so that $\langle v_1, \ldots, v_n \rangle = \langle \boldsymbol{z}, \boldsymbol{y}, \boldsymbol{x} \rangle$ and call i the *level* of variable v_i. A variable v_i is the *top variable* in I, if v_i is the largest variable in I with respect to $\prec$, and we denote with $\text{LEVEL}(I)$ the function that returns the level i. Given a set of inequalities $\mathcal{I}$, we can partition it by level and we denote with $\mathcal{I}_v$ the set of all inequalities from $\mathcal{I}$ with v as the top variable.

The algorithm maintains an assignment σ of variables to values in $\mathbb{Q}$. Any inequality I with v_i as the top variable implies a bound on the possible values that v_i can take with respect to the current assignment of $v_1, \ldots, v_{i-1}$. For example, if $I \equiv (v_i + p \leq 0)$, then the implied bound is $v_i \leq -\sigma(p)$. For each variable v_i, the algorithm maintains an interval $\text{FEASIBLE}[v_i] = (l, u)$ that represents the strongest lower and upper bounds inferred on v_i. Additionally, the bounds of this interval are associated with the inequalities $I_l[v_i], I_u[v_i]$ that imply them. We say that the variable v_i is in conflict, denoted with $\text{IN-CONFLICT}(v_i)$, if the current lower and upper bounds on v_i are in conflict, i.e., when $\text{FEASIBLE}[v_i]$ is either a (half-)open interval with $l \geq u$, or a closed interval with $l > u$.

The main inference mechanism in the algorithm is *bound propagation* on inequalities, which is an arithmetic analogue to unit propagation that SAT solvers perform on clauses. Given an inequality I with v_i as its top variable, we denote with $\text{PROPAGATE-BOUNDS}(I, v_i)$ the procedure that computes the bound that I implies on v_i and updates the bound information if the new bound is stronger than the existing one. We overload bound propagation to operate over a set of inequalities $\mathcal{I}_{v_i}$ with v_i as its top variable, and denote with $\text{PROPAGATE-BOUNDS}(\mathcal{I}_{v_i}, v_i)$ the procedure that resets the current bound information on v_i and then updates it by propagating bounds over all inequalities in $\mathcal{I}_{v_i}$. If, after performing exhaustive propagation over $\mathcal{I}_{v_i}$, the variable v_i is not in conflict, then we can safely pick a value $\alpha \in \text{FEASIBLE}[v_i]$, which we denote with $\text{PICK-VALUE}(v_i)$.[10] In this case, by construction, it is guaranteed that the value can be used to satisfy $\mathcal{I}_{v_i}$, i.e., $\sigma\{v_i \mapsto \alpha\} \vDash \mathcal{I}_{v_i}$.

The algorithm starts at level 1 and tries to gradually build a satisfying assignment σ for the variables $\boldsymbol{v}$, by assigning them values one by one. We know

[10] For example we can pick $\frac{l+u}{2}$, which is what we do in our implementation.

Algorithm 1 Interpolation with Conflict Resolution.

Require: Sets of inequalities $C_A(\boldsymbol{x}, \boldsymbol{y})$ and $C_B(\boldsymbol{y}, \boldsymbol{z})$, known to be inconsistent.
Ensure: Set of inequalities J is an interpolant for C_A and C_B.

```
1  function P_CR(C_A, C_B)
2      v ← ⟨z, y, x⟩                                   ▷ order the variables z ≺ y < x.
3      i ← 1; 𝓘 ← C_A ∪ C_B; J ← ∅                     ▷ initialize and start from bottom
4      loop
5          PROPAGATE-BOUNDS(𝓘_{v_i}, v_i)               ▷ compute bounds for v_i
6          while IN-CONFLICT(v_i) do                    ▷ resolve any conflicts
7              R ← FM-RESOLVE(I_l[v_i], I_u[v_i], v_i)  ▷ compute the resolvent
8              J ← J ∪ SELECT(R, I_l[v_i], I_u[v_i])    ▷ add relevant inequalities to J
9              if (R ≠ ⊥) then                          ▷ backtrack with resolvent
10                 i = LEVEL(R)                         ▷ level to backtrack to
11                 𝓘 = 𝓘 ∪ {R}                          ▷ remember the new inequality
12                 PROPAGATE-BOUNDS(R, v_i)             ▷ update bounds with new inequality
13             else return J                            ▷ J is the interpolant.
14         σ[v_i] ← PICK-VALUE(v_i)                     ▷ pick a value for v_i in FEASIBLE(v_i)
15         i ← i + 1                                    ▷ continue with next variable
```

that a complete model does not exist, and the failed model-building attempts will guide the process of FM resolution. At each level i the algorithm performs bound propagation to compute the interval of potential values that the variable v_i can take with respect to the assignment of variables $v_1, \ldots, v_{i-1}$. If bound propagation produces a feasible interval, then the algorithm assigns to v_i a value in this interval and moves on to the next variable. Otherwise, IN-CONFLICT(v_i) is true, and there are two inequalities[11]

$$I_l[v_i] \equiv (p - v_i < 0) \ , \qquad\qquad I_u[v_i] \equiv (v_i - q < 0) \ ,$$

such that the bounds they imply on v_i are inconsistent, i.e., we know that $\sigma(p) \geq \sigma(q)$. Mimicking a SAT solver, we can resolve this conflict by applying Fourier-Motzkin resolution to derive the resolvent $R = (p - q < 0)$. We denote the resolution inference over inequalities I_1 and I_2 that eliminates variable v_i with $R = \text{FM-RESOLVE}(I_1, I_2, v_i)$. The inequality R is a potential new node in the FM proof, and we examine the proof inference and add any relevant inequalities to the interpolant J. In addition we use the resolvent R to backtrack as follows. Since the resolvent R does not include v_i, it must be of level less than i. We also know by $\sigma(p) - \sigma(q) \geq 0$ that R is inconsistent with the current model, i.e. that $\sigma \not\models R$. If $R \equiv \bot$, we have found the proof of unsatisfiability and the set of inequalities J is the final interpolant. Otherwise, we use R to backtrack to the level of R and update the bounds of its top variable with new information.

Properties of P_{CR}. The termination and correctness of the algorithm follows from the termination and correctness of the original conflict resolution algorithm [24], and the fact that we can obtain the interpolant from the computed FM proof.

[11] For simplicity we only consider the case of strict inequalities, other cases are similar.

Another way of looking at the P_{CR} algorithm is as a semantic interpolation game where each model that can be constructed for C_B inequalities is refuted by an inequality derived from C_A.

FM elimination allows us to put a bound on the number of literals that can appear in the interpolant J. First, for a fixed global A, and any lemma interpolation problem (C_A, C_B), we know that $\textsc{literals}(C_A) \subseteq \textsc{literals}(A)$. Therefore, the inequalities that can appear in J are limited to the inequalities that one can obtain by FM elimination on $\textsc{literals}(A)$. From this bound and Lemma 2 we can then show the following two properties of P_{CR}.

Lemma 3. *P_{CR} is a finite-covering $\mathcal{T}_{\text{LA}}$-lemma interpolator.*

Lemma 4. $\textsc{itp}[\![P_{\text{CR}}]\!]$ *interpolation procedure has the finite convergence property.*

5 Improving Interpolation with Abstract Interpretation

The P_{CR} lemma interpolator described in the previous section ensures finite convergence of interpolation sequences. This is achieved by restricting the language of the potential interpolants by relying on quantifier elimination. Since our interest in interpolation comes from its use in construction of invariants, this also restricts the potential invariants that we can construct and can be seen as a disadvantage of the method. In this section we consider the interpolation problem specifically in the context of model checking and invariant inference and try to remedy this.

Model Checking. We assume a finite set of variables $\boldsymbol{x}$ called state variables. To each variable $x \in \boldsymbol{x}$, we associate its primed version x'. We call any formula $F(\boldsymbol{x})$ over the state variables a state formula, and any formula $T(\boldsymbol{x}, \boldsymbol{x}')$ a state-transition formula. A state-transition system is a pair $\mathfrak{S} = \langle I, T \rangle$, where $I(\boldsymbol{x})$ is a state formula describing the initial states and $T(\boldsymbol{x}, \boldsymbol{x}')$ is a state-transition formula describing the system's evolution.

Example 5. Let $\mathfrak{S} = \langle I, T \rangle$ be a transition system defined as

$$I \equiv (x = 0) \land (y = 0) \ , \qquad\qquad T \equiv (x' = x + 1) \land (y' = y + 1) \ .$$

It is easy to see that $(x = y)$ is an invariant of $\mathfrak{S}$. Nevertheless, consider a typical query that a model checker would use to check if a potential bad state $(x < y)$ is reachable.

$$\overbrace{I(x, y) \land T(x, y, x', y')}^{C_A} \land \overbrace{(x' < y')}^{C_B} \ .$$

The query above is unsatisfiable, and we can use it to derive an interpolant to use in invariant inference. Unfortunately, by using P_{CR}, since we are only inferring inequalities from C_A, we can never deduce an inequality that relates the variables x and y, and the resulting interpolant is $(x' = 1) \land (y' = 1)$.

On the other hand, because it must produce a single constraint, the Farkas-based P_{FK} will relate the variables x and y, and produce the desired interpolant as follows

$$(x' = x + 1) + (y' = y + 1) + (x = 0) + (y = 0) \equiv (x' = y') \ .$$

This remarkable capacity of P_{FK} to relate relevant variables is probably one of the main reasons for its successful adoption. □

Abstract Interpretation. As the example above shows, restricting the language of interpolants with quantifier elimination can put P_{CR} at a loss when inferring invariants. In order to improve the invariant-inference capacity of P_{CR}, we will rely on the tools provided by one of the most successful frameworks for invariant inference – abstract interpretation [12]. Abstract interpretation is a theory for sound approximation of the semantics of transition systems. The appeal of abstract interpretation is that it can efficiently compute a superset of all possible behaviors of a system by means of abstractions. These abstractions are computed by abstracting the semantics of the system with semantic techniques that are orthogonal to the syntactic, proof-based approach of quantifier elimination. Abstract interpretation provides a range of abstract domains $\mathcal{D}^{\#}$ that can be used for approximating $\mathcal{T}_{\text{LA}}$ transition systems, such as the interval [11], octagon [31], and the polyhedra [13] domains.

Since we are working in the context of model checking, we assume a global interpolation problem of the form

$$A(\boldsymbol{x}, \boldsymbol{x}') \equiv (I(\boldsymbol{x}) \wedge T(\boldsymbol{x}, \boldsymbol{x}')) \ , \qquad\qquad B(\boldsymbol{x}') \ ,$$

with $A \wedge B$ unsatisfiable. Note that the usual SMT interpolation procedures *do not have this information* (the relationship between $\boldsymbol{x}$ and $\boldsymbol{x}'$ variables has to be provided by the model checker).

As part of the proof of unsatisfiability of $A \wedge B$, we also assume a $\mathcal{T}_{\text{LA}}$-conflict separated into $C_A(\boldsymbol{x}, \boldsymbol{x}')$ and $C_B(\boldsymbol{x}')$ that we need to interpolate. The formula $C_A \wedge C_B$ is unsatisfiable and we can view C_A as containing one piece of the transition relation T. The transition piece itself is in convenient conjunctive form expressed with linear inequalities and we therefore pick the polyhedra domain as our precise concrete domain $\mathcal{C}^{\#}$. For the abstract domain we can choose any other arithmetic domain $\mathcal{D}^{\#}$ mentioned above. We will be using the following operations provided by the domains:

- Abstraction function $\alpha : \mathcal{C}^{\#} \mapsto \mathcal{D}^{\#}$, mapping concrete domain elements to their abstract representation.
- Concretization function $\gamma : \mathcal{D}^{\#} \mapsto \mathcal{C}^{\#}$, mapping abstract domain elements to their concrete representation.
- Join operator $\sqcup^{\#} : \mathcal{D}^{\#} \times \mathcal{D}^{\#} \mapsto \mathcal{D}^{\#}$ that, given two abstract domain elements, computes the abstract element capturing both of them.
- Projection operator $\exists^{\#}$ that can eliminate variables from elements of $\mathcal{D}^{\#}$.

Example 6. We illustrate the approach on Example 5 where P_{CR} was not satisfactory. Let $\mathcal{D}^{\#}$ be the polyhedra domain (so no abstraction is necessary). First, we project the C_A formula to its state representation and next state projections, obtaining

$$D_A = (\exists^{\#}\, x', y' \,.\, A) = (x = 0) \wedge (y = 0) \ ,$$
$$D_B = (\exists^{\#}\, x, y \,.\, A) = (x' = 1) \wedge (y' = 1) \ .$$

As usual in abstract interpretation, we can combine the two domain elements using a join operation to obtain

$$D_J = (D_A\{x, y/x', y'\}) \sqcup^{\#} D_B = (x' = y') \wedge (0 \leq x) \wedge (x \leq 1) \ ,$$

which is the invariant we were looking for.[12] □

As the example above shows, we can use the tools from abstract interpretation to infer new facts that take into account the transition system semantics (at least partially). In general, the facts inferred by abstract interpretation will not constitute an interpolant (they might not be inconsistent with C_B). But, the inferred facts are valid consequences of C_A, so we can freely conjoin them to C_A and resort to P_{CR} to complete the interpolant. The $\mathcal{T}_{\text{LA}}$-lemma interpolator P_{AI}, based on this approach, is fully described in Algorithm 2. We emphasize again that this approach relies on the model checker to provide the information about the transition system – the substitution $D_A\{\boldsymbol{x}/\boldsymbol{x}'\}$ at line 5 can not be done without knowing the correspondence between the $\boldsymbol{x}$ and $\boldsymbol{x}'$ variables.

Algorithm 2 Interpolation with Abstract Interpreation.

Require: Sets of inequalities $C_A(\boldsymbol{x}, \boldsymbol{x}')$ and $C_B(\boldsymbol{x}')$, known to be inconsistent.

1 **function** $P_{\text{AI}}(C_A, C_B)$
2 $\quad D \leftarrow \alpha(C_A)$ ▷ abstract the partial transition
3 $\quad D_A \leftarrow \exists^{\#}\boldsymbol{x}' .\, D$ ▷ project on state variables $\boldsymbol{x}$
4 $\quad D_B \leftarrow \exists^{\#}\boldsymbol{x} \,.\, D$ ▷ project on next-state variables $\boldsymbol{x}'$
5 $\quad D_J \leftarrow (D_A\{\boldsymbol{x}/\boldsymbol{x}'\}) \sqcup^{\#} D_B$ ▷ join the two abstractions
6 $\quad$ **return** $P_{\text{CR}}(C_A \cup \gamma(D_J), C_B)$ ▷ compute the interpolant

Properties of P_{AI}*.* The argument to show that P_{AI} is finite covering is similar to the argument we used for P_{CR}. For a fixed A, and any $\mathcal{T}_{\text{LA}}$-lemma interpolation problem (C_A, C_B), we know that $\text{LITERALS}(C_A) \subseteq \text{LITERALS}(A)$. Therefore, overall, abstract interpretation will always operate on subsets of $\text{LITERALS}(A)$ and can only ever infer a bounded number of new facts. This finite set of potential new literals does not interfere with finite covering by adding them to P_{CR}, it only increases the basis which quantifier elimination can derive inequalities from.

[12] For readers unfamiliar with the polyhedra domain, the join D_J is computed as the convex closure of the points $\{(0,0), (1,1)\}$.

Lemma 5. *P_{AI} is a finite-covering $\mathcal{T}_{LA}$-lemma interpolator.*

Lemma 6. ITP$[\![P_{AI}]\!]$ *interpolation procedure has the finite convergence property.*

6 Experiments

We have implemented the new interpolation method in the SALLY model-checker by relying on the MATHSAT5 SMT solver [9] for interpolation and APRON [20] for abstract interpretation over arithmetic domains.[13] We use the default PDKIND implementation in SALLY and denote with PDKIND+CR the method that uses the new interpolation method with conflict resolution (but no abstract interpretation), and with PDKIND+CR+POLKA the method that uses the new interpolation method with both conflict resolution and abstract interpretation based on the polyhedra domain.

We have evaluated the new procedure on a range of benchmarks. Several of our benchmarks are related to fault-tolerant algorithms (`om`, `ttesynchro` and `ttastartup`, `unifapprox`, `azadmanesh`, `approxagree`, `hacms`, and `misc` problem sets). We also used benchmarks from software model checking (`cav12`, `ctigar`). The `lustre` benchmarks are from the benchmark suite of the KIND model-checker, `cons` are simple concurrent programs, and `lfht` problems model a lock-free hash table. Some of the benchmarks were obtained from an existing repository.[14]

Our main goal is to illustrate the impact of the new interpolation method but, to put the results in context, we also compare NUXMV [5,8] (NUXMV was the most robust model checker in our previous work [22]) The results are presented in Figure 1. Each problem instance was run with a timeout of 10 minutes. Each column of the table corresponds to PDKIND with a different $\mathcal{T}_{LA}$-lemma interpolator, and each row corresponds to a different problem set. For each problem set and interpolator we report the number of problems that the tool has solved, how many of the solved problems were valid and invalid properties, and the total time (in seconds) that the tool took to solve those problems.

First we evaluate the impact of using conflict resolution (PDKIND+CR) and abstract interpretation (PDKIND+CR+POLKA) as the interpolator, compared to the default PDKIND with the Farkas-based interpolator. The results are shown in Figure 1. Overall, by adding conflict resolution as the lemma interpolator (PDKIND+CR), the method can find more counter-examples (but can prove fewer properties) than the default PDKIND. Then, by extending it with abstract interpretation (PDKIND+CR+POLKA), the tool retains the advantage at finding counter examples, but can, in addition, prove more valid properties. These results are aligned with our expectations. With the interpolation providing convergence guarantees, the PDKIND method does not get stuck in individual invariant

[13] SALLY is open source on GitHub. The majority of the interpolator code can be seen in `https://github.com/SRI-CSL/sally/blob/interpolation/src/smt/mathsat5/conflict_resolution.cpp`.

[14] `https://es-static.fbk.eu/people/griggio/vtsa2015/`

Fig. 1. Comparison of different $\mathcal{T}_{\text{LA}}$-lemma interpolators. Rows correspond to different problem sets and columns correspond to PDKIND with different interpolators (separate column for NUXMV for context). Each table entry shows the number of problems that the tool has solved, how many of those were valid and invalid, and the total time it took for the solved instances.

	PDKIND			PDKIND+CR			PDKIND+CR+POLKA			NUXMV		
problem set	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)
approxagree (9)	**9**	**8/1**	**185**	9	8/1	240	9	8/1	238	6	5/1	477
azadmanesh (20)	20	17/3	278	**20**	**17/3**	**173**	20	17/3	174	20	17/3	1269
cav12 (99)	68	48/20	2541	71	49/22	3722	**71**	**49/22**	**2966**	74	51/23	2910
conc (6)	4	4/0	117	3	3/0	6	**5**	**5/0**	**30**	4	4/0	220
ctigar (110)	**74**	**54/20**	**1252**	73	53/20	1532	74	54/20	1686	81	61/20	1829
hacms (5)	4	2/2	955	3	2/1	463	**5**	**3/2**	**923**	4	2/2	459
lfht (27)	17	17/0	106	17	17/0	681	**23**	**23/0**	**2194**	24	24/0	562
lustre (790)	772	438/334	2730	755	419/336	5209	**773**	**438/335**	**1964**	769	434/335	3542
misc (10)	8	7/1	117	8	6/2	200	**9**	**7/2**	**241**	8	8/0	320
om (9)	9	7/2	3	**9**	**7/2**	**1**	**9**	**7/2**	**1**	9	7/2	469
ttastartup (3)	1	1/0	7	1	1/0	7	**1**	**1/0**	**6**	1	1/0	1
ttesynchro (6)	6	3/3	16	**6**	**3/3**	**9**	**6**	**3/3**	**9**	5	2/3	1428
unifapprox (11)	11	8/3	225	11	8/3	134	**11**	**8/3**	**132**	11	8/3	271
	1003	614/389	8532	986	593/393	12377	**1016**	**623/393**	**10564**	1016	624/392	13757

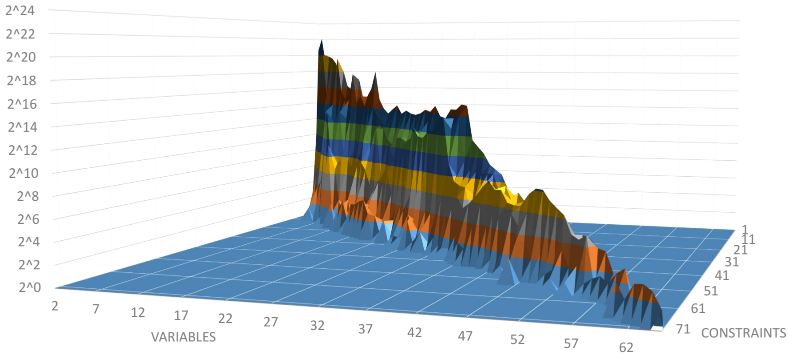
inference frames and can make progress towards the counter-examples. But, due to its restricted interpolation language it is bound to also be restricted in invariant inference, which is why PDKIND+CR can show fewer valid properties correct. The addition of abstract interpretation inferences improves this situation by extending the expressiveness of interpolants and making the interpolants more invariant-directed.

Next, we evaluate the effect of using different abstract domains. The APRON library provides the standard interval [11], octagon [31], and polyhedra [13] domains, and we denote variants of PDKIND that use these domains as PDKIND+CR+BOX, PDKIND+CR+OCT, and PDKIND+CR+POLKA. Results of the comparison are presented in Figure 2. The main takeaway from comparing different abstract domains is that, the more expressive the domain, the better the results. In general, expressive abstract domains are desirable in verification application, but suffer from scalability problems. The polyhedra domain, for example, is worst case exponential complexity in both space and time, and there is ongoing work to try and make it more efficient in practice [36]. In our context, we can easily apply the polyhedra domain even by relying on an off-the-shelf library such as APRON, as we use the domains solely on the cores of the theory lemmas produced by the SMT solver, where the number of variables and lemmas tends to be small. Figure 3 shows the distribution of the interpolation problems with respect to the number of constrains and the number of variables.

Fig. 2. Comparison of different abstract domains. Each row corresponds to a different problem set. Each column corresponds to PDKIND+CR with a different abstract domain. Each table entry shows the number of problems that the engine solved, how many of those were valid and invalid, and the total time it took for the solved instances.

	PDKIND+CR			PDKIND+CR+BOX			PDKIND+CR+OCT			PDKIND+CR+POLKA		
problem set	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)
approxagree (9)	9	8/1	240	9	8/1	240	**9**	**8/1**	**237**	9	8/1	238
azadmanesh (20)	20	17/3	173	**20**	**17/3**	**170**	**20**	**17/3**	**170**	20	17/3	174
cav12 (99)	71	49/22	3722	71	49/22	3291	67	48/19	1474	**71**	**49/22**	**2966**
conc (6)	3	3/0	6	5	5/0	35	5	5/0	47	**5**	**5/0**	**30**
ctigar (110)	73	53/20	1532	**74**	**54/20**	**1395**	73	53/20	884	74	54/20	1686
hacms (5)	3	2/1	463	4	2/2	799	4	3/1	1036	**5**	**3/2**	**923**
lfht (27)	17	17/0	681	20	20/0	1020	20	20/0	772	**23**	**23/0**	**2194**
lustre (790)	755	419/336	5209	757	421/336	3075	762	428/334	3063	**773**	**438/335**	**1964**
misc (10)	8	6/2	200	9	7/2	303	**9**	**7/2**	**220**	9	7/2	241
om (9)	**9**	**7/2**	**1**	**9**	**7/2**	**1**	**9**	**7/2**	**1**	**9**	**7/2**	**1**
ttastartup (3)	1	1/0	7	1	1/0	6	**2**	**1/1**	**459**	1	1/0	6
ttesynchro (6)	**6**	**3/3**	**9**	6	3/3	10	**6**	**3/3**	**9**	**6**	**3/3**	**9**
unifapprox (11)	11	8/3	134	11	8/3	131	**11**	**8/3**	**130**	11	8/3	132
	986	593/393	12377	996	602/394	10476	997	608/389	8502	**1016**	**623/393**	**10564**

Fig. 3. The distribution of the number of constraints and variables in $\mathcal{T}_{\text{LA}}$-lemma interpolation problems over our whole dataset (vertical axis is logarithmic). Of all interpolation problems, 66.55% have 5 variables or less, 87.18% have 10 variables or less, and 96.22% have 20 variables or less.



7 Conclusion

We presented a new approach for proof-based interpolation in SMT that can guarantee convergence of interpolation sequences and is invariant-driven. Both of these properties are valuable in the context of model checking techniques such as IC3/PDR. For example, with the new interpolation method, we can finally guarantee that the theoretical results for our own PDKIND method [22] also hold in practice. The approach combines two orthogonal approaches to invariant inference – symbolic reasoning through quantifier elimination and semantic reasoning with abstract interpretation – to provide interpolants that are both expressive invariant-driven facts and can be controlled to provide convergence guarantees. The new interpolation method takes advantage of the strengths of the individual parts of the usual model-checking reasoning stack. For a system under analysis, the model checker provides information about the system, the SMT solver discharges the control-flow of the system, the interpolator provides the symbolic forward reasoning, and the abstract interpretation improves the interpolants by interpreting the pieces of the system with no control flow. For example, both quantifier elimination and abstract interpretation can be expensive or ineffective on expressive domains when the problems involve many variables and disjunctions. Our new method sidesteps these issues since we only need to reason on the unsatisfiable cores provided by the SMT solver, which are minimal and conjunctive. The overall architecture of the new approach is shown in Figure 4.

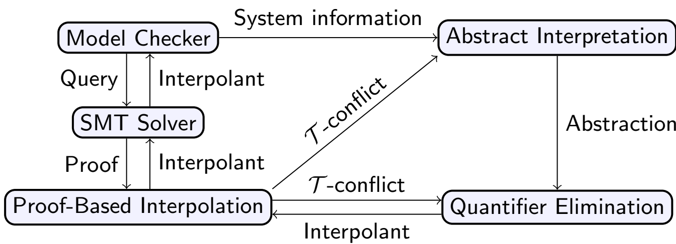


Fig. 4. All participants in the interpolation framework.

The method is implemented in the SALLY model checker, by relying on the proof-based interpolation framework of the MATHSAT5 SMT solver and the APRON abstract domain library. Our experimental evaluation shows that the new interpolation method is effective and improves the performance of SALLY in both invariant inference and bug-finding.

Future Work. The new interpolation method is modular and enables cross-pollination of the different techniques across the whole reasoning stack. This gives rise to many interesting directions for future work. As the first steps, we

plan to explore the use of tools from abstract interpretation to improve the interpolation in the theories of integer arithmetic, bit-vectors and arrays. In the other direction, we also see a possibility to contribute symbolic techniques to abstract interpretation. For example, if we lift the restriction that the lemma interpolants must refute the B part of the interpolation problem, the result of the proof-based interpolant computation is not an interpolant but rather an abstraction of the A formula. This could be a potential direction toward a property-driven logical interpretation (e.g., [37,3]). It is important to note that, although our new method guarantees convergence of the interplant sequences, it does not guarantee the convergence of the overall model-checking procedure. The overall convergence can be achieved by adding even more control over the interpolation language (e.g. [21]), and we plan to explore this direction.

Acknowledgements. We would like to thank Alberto Griggio for providing an interface to MATHSAT5 that allowed us to replace the default $\mathcal{T}$-interpolator with our custom external interpolator.

References

1. A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *International Conference on Computer Aided Verification*, pages 313–329. Springer, 2013.
2. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.
3. N. Bjørner and A. Gurfinkel. Property directed polyhedral abstraction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 263–281. Springer, 2015.
4. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *International Static Analysis Symposium*, pages 105–125. Springer, 2013.
5. R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.
6. J. Christ, J. Hoenicke, and A. Nutz. Proof tree preserving interpolation. In *TACAS*, volume 13, pages 124–138. Springer, 2013.
7. A. Cimatti and A. Griggio. Software model checking via IC3. In *CAV*, volume 7358, pages 277–293. Springer, 2012.
8. A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Tacas*, volume 8413, pages 46–61, 2014.
9. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795, pages 93–107. Springer, 2013.
10. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic (TOCL)*, 12(1):7, 2010.
11. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

12. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.
14. W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.
15. L. De Moura and D. Jovanović. A model-constructing satisfiability calculus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer, 2013.
16. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *VMCAI*, volume 10, pages 129–145. Springer, 2010.
17. B. Dutertre and L. De Moura. A fast linear-arithmetic solver for dpll (t). In *Computer Aided Verification*, pages 81–94. Springer, 2006.
18. K. Hoder, L. Kovacs, and A. Voronkov. Playing in the grey area of proofs. In *ACM SIGPLAN Notices*, volume 47, pages 259–272. ACM, 2012.
19. G. Huang. Constructing Craig interpolation formulas. *Computing and Combinatorics*, pages 181–190, 1995.
20. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009.
21. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 459–473. Springer, 2006.
22. D. Jovanović and B. Dutertre. Property-directed k-induction. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, pages 85–92. FMCAD Inc, 2016.
23. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.
24. K. Korovin, N. Tsiskaridze, and A. Voronkov. Conflict resolution. *Principles and Practice of Constraint Programming-CP 2009*, pages 509–523, 2009.
25. J. Krajíček. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *The Journal of Symbolic Logic*, 62(2):457–486, 1997.
26. K. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing dpll to richer logics. In *Computer Aided Verification*, pages 462–476. Springer, 2009.
27. K. L. McMillan. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.
28. K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.
29. K. L. McMillan. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*, pages 123–136. Springer, 2006.
30. K. L. McMillan. Interpolation: Proofs in the service of model checking. In *Hanbook of Model-Checking*. Springer, 2014.
31. A. Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.
32. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

33. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *The Journal of Symbolic Logic*, 62(3):981–998, 1997.
34. S. Rollini, O. Sery, and N. Sharygina. Leveraging interpolant strength in model checking. In *Computer Aided Verification*, pages 193–209. Springer, 2012.
35. A. Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1998.
36. G. Singh, M. Püschel, and M. T. Vechev. Fast polyhedra abstract domain. In *POPL*, pages 46–59, 2017.
37. A. Tiwari and S. Gulwani. Logical interpretation: Static program analysis using theorem proving. In *CADE*, volume 4603, pages 147–166. Springer, 2007.
38. G. Weissenbacher. Interpolant strength revisited. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 312–326. Springer, 2012.

An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs

Laura Titolo[1], Marco A. Feliú[1], Mariano Moscato[1]⋆, and César A. Muñoz[2]

[1] National Institute of Aerospace,
{laura.titolo, marco.feliu, mariano.moscato}@nianet.org
[2] NASA Langley Research Center,
cesar.a.munoz@nasa.gov

Abstract. This paper presents an abstract interpretation framework for the round-off error analysis of floating-point programs. This framework defines a parametric abstract analysis that computes, for each combination of ideal and floating-point execution path of the program, a sound over-approximation of the accumulated floating-point round-off error that may occur. In addition, a Boolean expression that characterizes the input values leading to the computed error approximation is also computed. An abstraction on the control flow of the program is proposed to mitigate the explosion of the number of elements generated by the analysis. Additionally, a widening operator is defined to ensure the convergence of recursive functions and loops. An instantiation of this framework is implemented in the prototype tool PRECiSA that generates formal proof certificates stating the correctness of the computed round-off errors.

1 Introduction

Floating-point numbers are often used as a finite representation of real numbers in computer programs. While floating-point numbers offer a good compromise between efficiency and precision for most applications, round-off errors in floating-point computations may be unacceptably large for some applications. In particular, in safety-critical systems, even small computational errors may have catastrophic consequences when they are not appropriately accounted for. To guarantee the safety of such systems, it is essential to correctly characterize the difference between a computed result and its ideal real number computation and the impact of this difference in the control-flow of a program.

Significant progress has been made in the last decade in the formal analysis of floating-point computations [1–6]. However, as stated in [7], none of the proposed approaches provides at the same time (i) a rigorous round-off errors analysis that generates externally checkable proofs certificates, (ii) the possibility of handling

⋆ Research by the first three authors was supported by the National Aeronautics and Space Administration under NASA/NIA Cooperative Agreement NNL09AA00A.

© Springer International Publishing AG 2018

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 516–537, 2018.
https://doi.org/10.1007/978-3-319-73721-8_24

a wide variety of mathematical operators, and (iii) sound support for typical programming language constructs such as conditionals, recursion, and loops. Another feature, which is not supported by the current errors analysis tools, is *compositionality*, i.e., the ability of analyzing a program in a modular way. This property is essential for obtaining a scalable and efficient approach.

This paper presents an abstract interpretation framework for the round-off error analysis of floating-point programs that addresses all the concerns above. The proposed framework defines a parametric semantics that collects, for each combination of ideal and floating-point computational path of a functional program, an error expression representing a provably sound upper-bound of the accumulated round-off error. Intuitively, the semantics associates conditions to each computed round-off error. The information accumulated in these conditions includes the path conditions, domain conditions ensuring that all expressions are total, e.g., divisors are non-zero, and additional conditions that enable tighter round-off errors for particular values. These conditions not only allow for more precise estimations of the round-off errors, but also enable the characterization of the input values that may lead to errors larger than expected.

The defined semantics is parametric with respect to round-off error bounds defined for a set of arithmetic operators. Hence, the analysis supports the extension of the programming language with new built-in operators as long as sound upper bounds of the operators' round-off errors are provided. The semantics is also parametric with respect to a set of execution paths of interests. These paths are individually examined by the analysis, while the other paths are condensed together in a sole abstract execution path. This abstraction makes the analysis more efficient and enables the analysis of programs with several nested conditionals. Finally, the semantics is parametric with respect to abstract domains of the real and Boolean expressions. Hence, the analysis supports different rigorous enclosure methods such as interval arithmetic, affine arithmetic, Bernstein and Taylor models, etc.

An instance of the presented framework has been implemented in the prototype tool PRECiSA. The input to PRECiSA is a functional program consisting of a set of floating-point functions. The output is a set of round-off error bounds with their associated conditions. Numerical values for these expressions are computed using an optimizer based on a formally verified branch-and-bound algorithm. PRECiSA generates proof certificates in the form of lemmas stating an accumulated round-off error estimation for each function in the program. These lemmas are equipped with proof scripts that automatically discharge them in an interactive theorem prover.

The paper is organized as follows. In Section 2, a formalization of floating-point round-off errors is presented. Section 3 presents the concrete semantics that computes the set of conditional error expressions associated to a program. In Section 4, the abstraction scheme and the widening operator are defined. A prototype tool that implements an instance of the proposed framework is presented in Section 5. Related work is discussed in Section 6. Section 7 concludes the paper.

2 Formalization of Floating-Point Round-off Errors

A floating-point number can be formalized as a pair of integers $(m, e) \in \mathbb{Z}^2$ [8,9] where m is called the *significand* and e the *exponent* of the float. A floating-point *format* f is defined as a pair of integers (p, e_{min}), where p is called the *precision* and e_{min} is called the *minimal exponent*. Given a base β, a pair $(m, e) \in \mathbb{Z}^2$ represents a floating-point number in the format (p, e_{min}) if and only if it holds that $|m| < \beta^p$ and $-e_{min} \leq e$. For instance, IEEE single and double precision floating-point numbers are specified by the formats $(24, 149)$ and $(53, 1074)$, respectively.

A conversion function $R : \mathbb{Z}^2 \to \mathbb{R}$ is defined to refer to the real number represented by a given float, i.e., $R((m, e)) = m \cdot \beta^e$. Since the function R is not injective, the representation of floating-point numbers is redundant. Therefore, notions about normality and canonicity are needed. A *canonical* float is a float such that is either a normal or subnormal. A *normal* float is a float such that the significand cannot be multiplied by the radix and still fit in the format. A *subnormal* is a float having the minimal exponent such that its significand can be multiplied by the radix and still fit in the format. Henceforth, $\mathbb{F}$ represents the set of floating-point numbers in canonical form. The expression $\tilde{v}$ will denote a floating-point number (m, e) in $\mathbb{F}$.

The expression $F_f(r)$ denotes the floating-point number in format f *closest* to r. The format f will be omitted when clear from the context. Let $\tilde{v}$ be a floating-point number that represents a real number r, the difference $|R(\tilde{v}) - r|$ is called the *round-off error* (or *rounding error*) of $\tilde{v}$ with respect to r. The *unit in the last place* (*ulp*) is a measure of the precision of a floating-point number as a representation of a real number. Given $r \in \mathbb{R}$, $ulp(r)$ represents the difference between two closest consecutive floating-point numbers $\tilde{v}_1$ and $\tilde{v}_2$ such that $\tilde{v}_1 \leq r \leq \tilde{v}_2$ and $\tilde{v}_1 \neq \tilde{v}_2$. It is defined in [9] as $ulp(\tilde{v}) = \beta^{e_{\tilde{v}}}$, where $e_{\tilde{v}}$ is the exponent of the *canonical* form of $\tilde{v}$ that is the floating-point number closest to r. The *ulp* can be used to bound the round-off error of a real number r with respect to its floating-point representation in the following way:

$$|R(F(r)) - r| \leq \tfrac{1}{2}\, ulp(r). \tag{2.1}$$

Given a set $\widetilde{\Omega}$ of pre-defined arithmetic floating-point operations, the corresponding set Ω of operations over real numbers, a denumerable set $\mathbb{V}$ of variables representing real values, and a denumerable set $\widetilde{\mathbb{V}}$ of variables representing floating-point values, where $\mathbb{V}$ and $\widetilde{\mathbb{V}}$ are disjoint, the sets $\mathbb{A}$ and $\widetilde{\mathbb{A}}$ of arithmetic expressions over real numbers and over floating-point numbers, respectively, are defined by the following grammar.

$$A ::= d \mid x \mid op(A, \ldots, A) \qquad \widetilde{A} ::= \tilde{d} \mid \tilde{x} \mid \widetilde{op}(\widetilde{A}, \ldots, \widetilde{A})$$

where $A \in \mathbb{A}$, $d \in \mathbb{R}$, $x \in \mathbb{V}$, $op \in \Omega$, $\widetilde{A} \in \widetilde{\mathbb{A}}$, $\tilde{d} \in \mathbb{F}$, $\tilde{x} \in \widetilde{\mathbb{V}}$, and $\widetilde{op} \in \widetilde{\Omega}$. It is assumed that there is a function $\chi_r : \widetilde{\mathbb{V}} \to \mathbb{V}$ that associates to each floating-point variable $\tilde{x}$ a variable $x \in \mathbb{V}$ representing the real value of $\tilde{x}$. Given a variable assignment $\sigma :$

$\mathbb{V} \to \mathbb{R}$, $eval_{\mathbb{A}}(\sigma, A) \in \mathbb{R}$ denotes the evaluation of the real arithmetic expression A with respect to σ. Similarly, given $\tilde{A} \in \widetilde{\mathbb{A}}$ and $\tilde{\sigma} : \widetilde{\mathbb{V}} \to \mathbb{F}$, $\widetilde{eval}_{\widetilde{\mathbb{A}}}(\tilde{\sigma}, \tilde{A}) \in \mathbb{F}$ denotes the evaluation of the floating-point arithmetic expression $\tilde{A}$ with respect to $\tilde{\sigma}$. The (partial) order relation between arithmetic expressions is defined as follows: $A_1 \leq A_2$ if and only if for all $\sigma : \mathbb{V} \to \mathbb{R}$, $eval_{\mathbb{A}}(\sigma, A_1) \leq eval_{\mathbb{A}}(\sigma, A_2)$.

The round-off error of the floating-point expression $\widetilde{op}(\tilde{v}_1, \ldots, \tilde{v}_n)$ with respect to the real-valued expression $op(r_1, \ldots, r_n)$, where $\widetilde{op}$ is a floating-point operator representing a real-valued operator op and $\tilde{v}_i$ is a floating-point value representing a real value r_i, for $1 \leq i \leq n$, depends of (a) the error introduced by the application of $\widetilde{op}$ versus op and (b) the propagation of the errors carried out by the arguments, i.e., the difference between $\tilde{v}_i$ and r_i, for $1 \leq i \leq n$, in the application. In the case of arithmetic operators, the IEEE-754 standard states that every basic operation is correctly rounded, therefore it should be performed as if it would be calculated with infinite precision and then rounded to the nearest floating-point value. Then, from Formula (2.1), the application of an n-ary floating-point operator $\widetilde{op}$ to the floating-point values $\tilde{v}_1, \ldots, \tilde{v}_n$ must fulfill the following condition.

$$|R(\widetilde{op}(\tilde{v}_i)_{i=1}^n) - op(R(\tilde{v}_i))_{i=1}^n| \leq \tfrac{1}{2}\, ulp(op(R(\tilde{v}_i))_{i=1}^n), \tag{2.2}$$

where the notation $f(x_i)_{i=1}^n$ is used to represent $f(x_1, \ldots, x_n)$.

To estimate how the errors of the arguments are propagated to the result of the application of the operator, it is necessary to bound the difference between the application of the real operator on real values and the application of the same operator on the floating-point arguments. The expression $\epsilon_{op}(e_i)_{i=1}^n$ is used to represent such difference, where each e_i is a bound of the round-off error carried by every floating-point $\tilde{v}_i$ representing a real value r_i, i.e., $|R(\tilde{v}_i) - r_i| \leq e_i$. Therefore, $\epsilon_{op}(e_i)_{i=1}^n$ satisfies the following condition.

$$|op(R(\tilde{v}_i))_{i=1}^n - op(r_i)_{i=1}^n| \leq \epsilon_{op}(e_i)_{i=1}^n. \tag{2.3}$$

The following bound of the round-off error between the floating-point expression and the real-valued counterpart follows from Formula (2.2), Formula (2.3), and the triangle inequality.

$$|R(\widetilde{op}(\tilde{v}_i)_{i=1}^n) - op(r_i)_{i=1}^n| \leq \epsilon_{op}(e_i)_{i=1}^n + \tfrac{1}{2}\, ulp(op(R(\tilde{v}_i))_{i=1}^n). \tag{2.4}$$

In this paper, for a given expression, the round-off error in the right-hand side of Formula (2.4) is expressed as an error expression.

Definition 1 (Error Expression). *An* error expression *is an arithmetic expression or the element $+\infty$ representing an arbitrary large round-off error.*

The domain of error expressions is denoted as $\mathbb{E}$ and it is defined as $\mathbb{E} := \mathbb{A} \cup \{+\infty\}$. The order relation on error expressions naturally extends the one on arithmetic expressions by stating that for all $e \in \mathbb{E}$, $e \leq +\infty$. The function *max* (respectively *min*) returns the maximum (respectively minimum) of a set error expressions with respect to the order relation $\leq$. The tuple $(\mathbb{E}, \leq, max, min, +\infty, 0)$ is a

complete lattice, where *max* is the least upper bound, *min* is the greatest lower bound, $+\infty$ is the greatest element of the domain, and 0 is the least element of the domain.

Additional conditions are needed in Formula (2.4) when the operators are not total. For example, when dealing with the division operation, it is necessary to guarantee that the second argument of both the floating-point operator and the real-valued operator is not zero. Furthermore, some arithmetic operations are associated to tighter error bounds under certain conditions. These conditions can be used to refine the estimation of the round-off error. Boolean expressions are used to model such conditions.

The sets $\mathbb{B}$ and $\widetilde{\mathbb{B}}$ of Boolean expressions over real numbers and over floating-point numbers, respectively, are defined by the following grammar.

$$B ::= true \mid false \mid B \wedge B \mid B \vee B \mid \neg B \mid A < A \mid A = A$$
$$\widetilde{B} ::= true \mid false \mid \widetilde{B} \wedge \widetilde{B} \mid \widetilde{B} \vee \widetilde{B} \mid \neg \widetilde{B} \mid \widetilde{A} < \widetilde{A} \mid \widetilde{A} = \widetilde{A}$$

where $B \in \mathbb{B}$, $A \in \mathbb{A}$, $\widetilde{B} \in \widetilde{\mathbb{B}}$, and $\widetilde{A} \in \widetilde{\mathbb{A}}$. The conjunction $\wedge$, disjunction $\vee$, negation $\neg$, *true*, and *false* have the usual classical logic meaning.

Given a variable assignment $\sigma : \mathbb{V} \to \mathbb{R}$, $eval_{\mathbb{B}}(\sigma, B) \in \{true, false\}$ denotes the evaluation of the real Boolean expression B. In the same way, given $\widetilde{B} \in \widetilde{\mathbb{B}}$ and $\widetilde{\sigma} : \widetilde{\mathbb{V}} \to \mathbb{F}$, $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\widetilde{\sigma}, \widetilde{B}) \in \{true, false\}$ denotes the evaluation of the floating-point Boolean expression $\widetilde{B}$. The (partial) order relation between Boolean expressions over real numbers is defined as follows: $B_1 \Rightarrow B_2$ if and only if for all $\sigma : \mathbb{V} \to \{true, false\}$, $eval_{\mathbb{B}}(\sigma, B_1)$ implies $eval_{\mathbb{B}}(\sigma, B_2)$. Similarly, for floating-point Boolean expressions, the order relation is defined as follows: $\widetilde{B}_1 \Rightarrow \widetilde{B}_2$ if and only if for all $\tilde{\sigma} : \widetilde{\mathbb{V}} \to \{true, false\}$, $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \widetilde{B}_1)$ implies $\widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma}, \widetilde{B}_2)$. The symbol *true* (respectively *false*) is the greatest (respectively least) Boolean expression of both domains $\mathbb{B}$ and $\widetilde{\mathbb{B}}$. The equivalence relation derived from $\Rightarrow$ is defined as $B_1 \Leftrightarrow B_2$ if and only if $B_1 \Rightarrow B_2$ and $B_2 \Rightarrow B_1$. In the following, by abuse of notation, a formula $B \in \mathbb{B} \cup \widetilde{\mathbb{B}}$ and its equivalence class will be denoted with the same symbol.

The function $R_{\mathbb{B}} : \widetilde{\mathbb{B}} \to \mathbb{B}$ that converts a Boolean expression on floating-point numbers to a Boolean expression on real numbers is defined by simply replacing each floating-point operation with the corresponding operation on real numbers and by applying R and χ_r to floating-point values and variables, respectively.

Henceforth, it is assumed that for any floating-point operator of interest *op* there exists at least one formula of the following form that holds for all $e_1, \ldots, e_n \in \mathbb{E}$ such that $|R(\tilde{v}_i) - r_i| \le e_i$ with $1 \le i \le n$,

$$\phi_{op}(r_i)_{i=1}^n \wedge \phi_{\widetilde{op}}(\tilde{v}_i)_{i=1}^n \text{ implies } |R(\widetilde{op}(\tilde{v}_i)_{i=1}^n) - op(r_i)_{i=1}^n| \le \epsilon_{\widetilde{op}}(r_i, e_i)_{i=1}^n, \quad (2.5)$$

where $\phi_{op}(r_i)_{i=1}^n \in \mathbb{B}$, $\phi_{op}(r_i)_{i=1}^n \not\Rightarrow false$, $\phi_{\widetilde{op}}(\tilde{v}_i)_{i=1}^n \in \widetilde{\mathbb{B}}$, $\phi_{\widetilde{op}}(\tilde{v}_i)_{i=1}^n \not\Rightarrow false$, and $\epsilon_{\widetilde{op}} : \mathbb{A}^n \times \mathbb{E}^n \to \mathbb{E}$. For the same floating-point operator there may be more than one formula of the form of Formula (2.5). In this case, the disjunction of all conditions in the left-hand side of Formula (2.5) should be complete for the domain of the operator. The framework presented in this paper does not require

those conditions to be disjoint, but better estimations are usually computed when these conditions are disjoint.

Example 1. Instances of Formula (2.5) for the four basic arithmetic operators are defined below.

- $\epsilon_{\tilde{+}}(r_1, e_1, r_2, e_2) := e_1 + e_2 + 1/2\, ulp(|r_1 + r_2| + e_1 + e_2)$, $\phi_{+}(r_1, r_2) := true$, and $\phi_{\tilde{+}}(\tilde{v}_1, \tilde{v}_2) := true$.
- $\epsilon_{\tilde{-}}(r_1, e_1, r_2, e_2) := e_1 + e_2 + 1/2\, ulp(|r_1 - r_2| + e_1 + e_2)$, $\phi_{-}(r_1, r_2) := true$, and $\phi_{\tilde{-}}(\tilde{v}_1, \tilde{v}_2) := \tilde{v}_2 \tilde{/} 2 > \tilde{v}_1 \vee \tilde{v}_1 > 2 \tilde{*} \tilde{v}_2$.
- $\epsilon_{\tilde{-}}(r_1, e_1, r_2, e_2) := e_1 + e_2$, $\phi_{-}(r_1, r_2) := true$ and $\phi_{\tilde{-}}(\tilde{v}_1, \tilde{v}_2) := \tilde{v}_2 \tilde{/} 2 \leq \tilde{v}_1 \wedge \tilde{v}_1 \leq 2 \tilde{*} \tilde{v}_2$.
- $\epsilon_{\tilde{*}}(r_1, e_1, r_2, e_2) := |r_1| e_2 + |r_2| e_1 + e_1 e_2 + 1/2\, ulp((|r_1| + e_1)(|r_2| + e_2))$, $\phi_{*}(r_1, r_2) :=$ $true$, and $\phi_{\tilde{*}}(\tilde{v}_1, \tilde{v}_2) := true$.
- $\epsilon_{\tilde{/}}(r_1, e_1, r_2, e_2) := \frac{|r_1| e_2 + |r_2| e_1}{r_2 r_2 - e_2 |r_2|} + 1/2\, ulp(\frac{|r_1| + e_1}{|r_2| - e_2})$, $\phi_{/}(r_1, r_2) := r_2 \neq 0$, and $\phi_{\tilde{/}}(\tilde{v}_1, \tilde{v}_2)$ $:= \tilde{v}_2 \neq 0$.

For instance, the round-off error of the sum includes the propagation of the errors of the operands (e_1 and e_2) and the error of rounding the result of the sum ($1/2\, ulp(|r_1 - r_2| + e_1 + e_2)$). In the case of the division operator, Boolean conditions are used to guarantee the validity of the operation, i.e., the conditions $\phi_{/}$ and $\phi_{\tilde{/}}$ state that the divisors of the real and floating point expressions, respectively, are different from zero. In the case of the subtraction operator, conditions that improve the error approximation are provided. Indeed, in [10], it is proven that the floating-point subtraction $x \tilde{-} y$ is computed exactly when $y \tilde{/} 2 \leq x \leq 2 \tilde{*} y$.

3 Concrete Denotational Semantics

This section presents a compositional structural denotational semantics for a generic declarative programming language. This semantics collects information about the round-off error of floating point operations and relies on the floating-point error formalization presented in Section 2. This semantics is an enhancement of the one introduced in [6] and it uses a more expressive domain.

The expression language considered in this paper contains conditionals, let expressions, and function calls, possibly recursive. Given a set $\widetilde{\Omega}$ of pre-defined arithmetic floating-point operations, a set Σ of function symbols, and a denumerable set $\widetilde{\mathbb{V}}$ of floating-point variables, $\mathbb{S}$ denotes the set of program expressions. The syntax of programs in $\mathbb{S}$ is given by the following grammar, where the syntax of floating-point arithmetic expressions given in Section 2 is augmented with a function call.

$$\widetilde{A} ::= \tilde{d} \mid F(d) \mid \tilde{x} \mid \widetilde{op}(\widetilde{A}, \ldots, \widetilde{A}) \mid\ f(\widetilde{A}, \ldots, \widetilde{A})$$
$$S ::= \widetilde{A} \mid if\ \widetilde{B}\ then\ S\ else\ S \mid let\ \tilde{x} = \widetilde{A}\ in\ S$$

where $\widetilde{A} \in \widetilde{\mathbb{A}}$, $\widetilde{B} \in \widetilde{\mathbb{B}}$, $\tilde{d} \in \mathbb{F}$, $d \in \mathbb{R}$, $\tilde{x} \in \widetilde{\mathbb{V}}$, $\widetilde{op} \in \widetilde{\Omega}$, and $f \in \Sigma$. Bounded recursion is added to the language as syntactic sugar using the convention $for(i, j, S, g) := if$ $i > j\ then\ S\ else\ g(j, for(i, j-1, S, g))$.

A program is defined as a set of *function declarations* of the form $f(\tilde{x}_1,\ldots,\tilde{x}_n) = S$, where $\tilde{x}_1,\ldots,\tilde{x}_n$ are pairwise distinct variables in $\widetilde{\mathbb{V}}$ and all free variables appearing in S are in $\{\tilde{x}_1,\ldots,\tilde{x}_n\}$. The natural number n is called the *arity* of f. Henceforth, it is assumed that programs are well-formed in the sense that for every function call $f(\tilde{x}_1,\ldots,\tilde{x}_n)$ that occurs in a program P, a unique function f of arity n is defined in P. The set of programs is denoted as $\mathbb{P}$.

The proposed semantics collects, for each program path, the corresponding path conditions (for both the real and the floating-point execution), and two expressions representing (1) the value of the output assuming the use of real arithmetic and (2) an upper bound for the accumulated round-off error that might affect the result due to floating-point operations. Since the semantics collects information about real and floating-point execution paths, it is possible to consider the error of taking the incorrect branch compared to the ideal execution using real arithmetic. This enables a sound treatment of unstable tests.

Definition 2 (Test Stability). *A conditional statement if $\tilde{\phi}$ then $\tilde{E}_1$ else $\tilde{E}_2$ is said to be* unstable *if there exist two assignments $\tilde{\sigma}:\widetilde{\mathbb{V}}\to\mathbb{F}$ and $\sigma:\mathbb{V}\to\mathbb{R}$ such that for all $\tilde{x}\in\widetilde{\mathbb{V}}$, $\sigma(\chi_r(\tilde{x})) = R(\tilde{\sigma}(\tilde{x}))$ and $eval_{\mathbb{B}}(\sigma, R_{\mathbb{B}}(\tilde{\phi})) \neq \widetilde{eval}_{\widetilde{\mathbb{B}}}(\tilde{\sigma},\tilde{\phi})$. Otherwise the conditional expression is said to be* stable.

In other words, a conditional statement is unstable when there exists an assignment from the variables in $\tilde{\phi}$ to $\mathbb{F}$ such that $\tilde{\phi}$ and $R_{\mathbb{B}}(\tilde{\phi})$ evaluate to different Boolean values.

A *condition* is a set of pairs of the form $(\phi,\tilde{\phi})$, with $\phi\in\mathbb{B}$ and $\tilde{\phi}\in\widetilde{\mathbb{B}}$. The domain of conditions is $(\wp(\mathbb{B}\times\widetilde{\mathbb{B}}), \hat{\Rightarrow}, \hat{\vee}, \hat{\wedge}, \{(true, true)\}, \{(false, false)\})$, where

- $\hat{\Rightarrow}$ is the order relation over $\wp(\mathbb{B}\times\widetilde{\mathbb{B}})$ defined as for all $\eta_1,\eta_2\in\wp(\mathbb{B}\times\widetilde{\mathbb{B}})$, $\eta_1 \hat{\Rightarrow} \eta_2$ if and only if $\bigvee_{(b_1,\tilde{b}_1)\in\eta_1}(b_1\wedge\tilde{b}_1) \Rightarrow \bigvee_{(b_2,\tilde{b}_2)\in\eta_2}(b_2\wedge\tilde{b}_2)$,
- the equivalence relation $\hat{\Leftrightarrow}$ derived from $\hat{\Rightarrow}$ is defined as follows, $\eta_1 \hat{\Leftrightarrow} \eta_2$ if and only if $\eta_1 \hat{\Rightarrow} \eta_2$ and $\eta_2 \hat{\Rightarrow} \eta_1$, and the equivalence class of a condition η is denoted as $[\eta]_{\hat{\Leftrightarrow}}$,
- $\hat{\vee}$ is the least upper bound defined as $\eta_1 \hat{\vee} \eta_2 = [\eta_1\cup\eta_2]_{\hat{\Leftrightarrow}}$,
- $\hat{\wedge}$ is the greatest lower bound defined as $\eta_1 \hat{\wedge} \eta_2 = \bigcup\{(b_1\wedge b_2, \tilde{b}_1\wedge\tilde{b}_2) \mid (b_1,\tilde{b}_1)\in\eta_1 (b_2,\tilde{b}_2)\in\eta_2\}$,
- $\{(true, true)\}$ is the greatest element of the domain, and
- $\{(false, false)\}$ is the least element of the domain.

Paths in the control flow of a program are represented by sequences, possibly empty, of 0's and 1's.

Definition 3 (Decision path). *A* decision path π *is defined by the grammar $\pi = \varepsilon \mid \pi\cdot 0 \mid \pi\cdot 1$, where ε denotes the empty path and $\cdot$ is the concatenation operator.*

The domain of all decision paths is denoted by *Path*. A decision path π models all the decision paths π' such that π is prefix of π'. Given $\pi_1,\pi_2\in Path$, the order relation on decision paths is defined as $\pi_1 \leq_{prefix} \pi_2$ if and only if

π_1 is a prefix of π_2. A decision path univocally identifies a subprogram or subexpression inside the input program. Subexpressions corresponding to the *then* branch of a conditional statement are identified by the index 1. Conversely, the subexpressions corresponding to the *else* branch are identified by the index 0. For example, consider the following program expression:

$$E = \mathit{if}\ \tilde{x} > 0\ \mathit{then}\ (\mathit{if}\ \tilde{y} > 2\ \mathit{then}\ 5\ \mathit{else}\ \tilde{y} + 1)\ \mathit{else}\ (\mathit{if}\ \tilde{z} > 0\ \mathit{then}\ \tilde{x} + \tilde{z}\ \mathit{else}\ \tilde{y} * \tilde{z})$$

All the decision paths of expression E are identified by ε. The path corresponding to the arithmetic expression $\tilde{y}+1$ is $1{\cdot}0$, and the path corresponding to expression $\tilde{x} + \tilde{z}$ is $0 \cdot 1$.

The semantics collects information in the form of *conditional error bounds.*

Definition 4 (Conditional Error Bound). *A* conditional error bound *is an expression of the form* $\langle \eta \rangle_t \twoheadrightarrow (r, e)^\pi$, *where* $\eta \in \wp(\mathbb{B} \times \widetilde{\mathbb{B}})$, $r \in \mathbb{A}$, $e \in \mathbb{E}$, $\pi \in \mathit{Path}$, *and* $t \in \{\mathbf{s}, \mathbf{u}\}$. *A conditional error bound is said to be* valid *if it exists* $(\phi, \tilde{\phi}) \in \eta$, $\phi \not\Rightarrow \mathit{false}$ *and* $\tilde{\phi} \not\Rightarrow \mathit{false}$.

Intuitively, $\langle \eta \rangle_t \twoheadrightarrow (r, e)^\pi$ indicates that for the decision path π, if the condition η is satisfied, the output of the ideal real numbers implementation of the program is r and the round-off error of the floating-point implementation is bounded by e. The sub-index t is used to mark by construction whether a conditional error bound is unstable ($t = \mathbf{u}$), or stable ($t = \mathbf{s}$).

Conditional error bounds are ordered in the following way $\langle \eta_1 \rangle_{t_1} \twoheadrightarrow (r_1, e_1)^{\pi_1} \leq \langle \eta_2 \rangle_{t_2} \twoheadrightarrow (r_2, e_2)^{\pi_2}$ if and only if $\eta_1 \hat{\Rightarrow} \eta_2$, $r_1 = r_2$, $e_1 \leq e_2$, $\pi_2 \leq_{\mathit{prefix}} \pi_1$, and $t_1 = t_2$. The domain $\mathbf{C}$ of conditional error bounds is defined as a set of tuples in $\wp(\mathbb{B} \times \widetilde{\mathbb{B}}) \times \mathbb{A} \times \mathbb{E} \times \mathit{Path} \times \{\mathbf{s}, \mathbf{u}\}$. Sets of conditional error bounds are (partially) ordered as follows. For all $C_1, C_2 \subseteq \mathbf{C}$, $C_1 \sqsubseteq C_2$ if and only iff for all $c_1 \in C_1$, there exists $c_2 \in C_2$ such that $c_1 \leq c_2$. The equivalence relation derived from $\sqsubseteq$ is defined as $C_1 \equiv C_2$ if and only if $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. In the following, by abuse of notation, the quotient of $\sqsubseteq$ over equivalence classes will be denoted with the same symbol. Furthermore, sets of conditional error bounds will be used modulo $\equiv$ and their class will be denoted as $\mathbb{C}$. The domain $(\mathbb{C}, \sqsubseteq, \bigsqcup, \bigsqcap, [\mathbf{C}]_\equiv, \varnothing)$ is a complete lattice where the least upper bound is defined as $C_1 \sqcup C_2 := [C_1 \cup C_2]_\equiv$ and the greatest lower bound is defined as $C_1 \sqcap C_2 := [\{c \in \mathbf{C} \mid \exists c_1 \in C_1 . c \leq c_1, \exists c_2 \in C_2 . c \leq c_2\}]_\equiv$.

An *environment* is defined as a function mapping a variable to a set of conditional error bounds, i.e., $\mathit{Env} = \widetilde{\mathbb{V}} \to \mathbb{C}$. The empty environment is denoted as $\bot_{\mathit{Env}}$ and maps every variable to the empty set $\varnothing$. Let $\mathbb{M} := \{f(\tilde{x}_1, \ldots, \tilde{x}_n) \mid f \in \Sigma, \tilde{x}_1, \ldots, \tilde{x}_n \in \widetilde{\mathbb{V}}\}$ be the set of all possible function calls. An *interpretation* is a function $I{:}\mathbb{M} \to \mathbb{C}$ modulo variance[3]. The set of all interpretations is denoted as $\mathbb{I}$. The empty interpretation is denoted as $\bot_{\mathbb{I}}$ and maps everything to $\varnothing$.

Let $\widetilde{op}$ be an n-ary floating-point operator in $\widetilde{\Omega}$ such that op in Ω is its real-valued counterpart and there exist $\epsilon_{\widetilde{op}} : \mathbb{A}^n \times \mathbb{E}^n \to \mathbb{E}$, $\phi_{op}(r_i)_{i=1}^n \in \mathbb{B}$ and

[3] Two functions $I_1, I_2{:}\mathbb{M} \to \mathbb{C}$ are variants if for each $m \in \mathbb{M}$ there exists a renaming ρ such that $(I_1(m))\rho = I_2(m\rho)$.

$\phi_{\widetilde{op}}(\tilde{v}_i)_{i=1}^n \in \widetilde{\mathbb{B}}$ such that Formula (2.5) holds. Given $\sigma \in Env$ and $I \in \mathbb{I}$, the semantics of program expressions, $\mathcal{E} : \mathbb{S} \times Env \times \mathbb{I} \times Path \to \mathbb{C}$, returns the set of conditional error bounds representing an upper bound of the round-off error for each execution path, together with the corresponding conditions. The function $\chi_e : \widetilde{\mathbb{V}} \to \mathbb{V}$ associates to each floating-point variable $\tilde{x}$ a variable in $\mathbb{V}$ representing the error of $\tilde{x}$. In the following, for the sake of simplicity, the singleton condition $\langle\{(\phi, \tilde{\phi})\}\rangle$ will be denoted as $\langle\phi, \tilde{\phi}\rangle$.

$$\mathcal{E}[\![\tilde{d}]\!]^{\pi}_{(\sigma,I)} := \{\langle true, true\rangle_{\mathbf{s}} \twoheadrightarrow (R(\tilde{d}), 0)^{\pi}\}$$

$$\mathcal{E}[\![F(d)]\!]^{\pi}_{(\sigma,I)} := \{\langle true, true\rangle_{\mathbf{s}} \twoheadrightarrow (d, |d - F(d)|)^{\pi}\}$$

$$\mathcal{E}[\![\tilde{x}]\!]^{\pi}_{(\sigma,I)} := \begin{cases} \{\langle true, true\rangle_{\mathbf{s}} \twoheadrightarrow (\chi_r(\tilde{x}), \chi_e(\tilde{x}))^{\pi}\} & \text{if } \sigma(\tilde{x}) = \varnothing \\ \sigma(\tilde{x}) & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![\widetilde{op}(\widetilde{A}_i)_{i=1}^n]\!]^{\pi}_{(\sigma,I)} :=$$

$$\bigsqcup\{\langle \bigwedge_{i=1}^{n} \phi_i \wedge \phi_{op}(r_i)_{i=1}^n, \bigwedge_{i=1}^{n} \tilde{\phi}_i \wedge \phi_{\widetilde{op}}(\widetilde{A}_i)_{i=1}^n\rangle_{\mathbf{s}} \twoheadrightarrow (op(r_i)_{i=1}^n, \epsilon_{\widetilde{op}}(r_i, e_i)_{i=1}^n)^{\pi} \mid \forall 1 \leq i \leq n:$$

$$\langle\phi_i, \tilde{\phi}_i\rangle_{\mathbf{s}} \twoheadrightarrow (r_i, e_i)^{\pi_i} \in \mathcal{E}[\![\widetilde{A}_i]\!]^{\pi}_{(\sigma,I)}, \bigwedge_{i=1}^{n} \phi_i \wedge \phi_{op}(r_i)_{i=1}^n \not\Rightarrow false, \bigwedge_{i=1}^{n} \tilde{\phi}_i \wedge \phi_{\widetilde{op}}(\widetilde{A}_i)_{i=1}^n \not\Rightarrow false\}$$

$$\mathcal{E}[\![let\ \tilde{x} = \widetilde{A}\ in\ S]\!]^{\pi}_{(\sigma,I)} := \mathcal{E}[\![S]\!]^{\pi}_{(\sigma[\tilde{x} \mapsto \mathcal{E}[\![\widetilde{A}]\!]^{\pi}_{(\sigma,I)}], I)}$$

$$\begin{aligned}
\mathcal{E}[\![if\ \widetilde{B}\ then\ S_1\ else\ S_2]\!]^{\pi}_{(\sigma,I)} :=\ & \mathcal{E}[\![S_1]\!]^{\pi\cdot 1}_{(\sigma,I)} \Downarrow_{(R_{\mathbb{B}}(\widetilde{B}), \widetilde{B})} \sqcup\ \mathcal{E}[\![S_2]\!]^{\pi\cdot 0}_{(\sigma,I)} \Downarrow_{(\neg R_{\mathbb{B}}(\widetilde{B}), \neg\widetilde{B})} \sqcup \\
& \bigsqcup\{\langle\phi_2, \tilde{\phi}_1\rangle_{\mathbf{u}} \twoheadrightarrow (r_2, e_1 + |r_1 - r_2|)^{\varepsilon} \mid \langle\phi_1, \tilde{\phi}_1\rangle_{t_1} \twoheadrightarrow (r_1, e_1)^{\pi_1} \in \mathcal{E}[\![S_1]\!]^{\pi\cdot 0}_{(\sigma,I)}, \\
& \qquad \langle\phi_2, \tilde{\phi}_2\rangle_{t_2} \twoheadrightarrow (r_2, e_2)^{\pi_2} \in \mathcal{E}[\![S_2]\!]^{\pi\cdot 1}_{(\sigma,I)}\} \Downarrow_{(\neg R_{\mathbb{B}}(\widetilde{B}), \widetilde{B})} \sqcup \\
& \bigsqcup\{\langle\phi_1, \tilde{\phi}_2\rangle_{\mathbf{u}} \twoheadrightarrow (r_1, e_2 + |r_1 - r_2|)^{\varepsilon} \mid \langle\phi_1, \tilde{\phi}_1\rangle_{t_1} \twoheadrightarrow (r_1, e_1)^{\pi_1} \in \mathcal{E}[\![S_1]\!]^{\pi\cdot 1}_{(\sigma,I)}, \\
& \qquad \langle\phi_2, \tilde{\phi}_2\rangle_{t_2} \twoheadrightarrow (r_2, e_2)^{\pi_2} \in \mathcal{E}[\![S_2]\!]^{\pi\cdot 0}_{(\sigma,I)}\} \Downarrow_{(R_{\mathbb{B}}(\widetilde{B}), \neg\widetilde{B})}
\end{aligned}$$

$$\mathcal{E}[\![f(\widetilde{A}_i)_{i=1}^n]\!]^{\pi}_{(\sigma,I)} := \bigsqcup\{\langle\phi' \wedge \bigwedge_{i=1}^{n} \phi_i, \tilde{\phi}' \wedge \bigwedge_{i=1}^{n} \tilde{\phi}_i\rangle_t \twoheadrightarrow (r', e')^{\pi'} \mid$$

$$\begin{aligned}
& \langle\phi, \tilde{\phi}\rangle_t \twoheadrightarrow (r, e)^{\pi'} \in I(f(\tilde{x}_i)_{i=1}^n), \forall 1 \leq i \leq n: \langle\phi_i, \tilde{\phi}_i\rangle_{t_i} \twoheadrightarrow (r_i, e_i)^{\pi_i} \in \mathcal{E}[\![\widetilde{A}_i]\!]^{\pi}_{(\sigma,I)}, \\
& r' = r[\chi_r(\tilde{x}_i)/r_i]_{i=1}^n,\ e' = e[\chi_e(\tilde{x}_i)/e_i]_{i=1}^n, \phi' = \phi[\chi_r(\tilde{x}_i)/r_i, \chi_e(\tilde{x}_i)/e_i]_{i=1}^n, \\
& \tilde{\phi}' = \tilde{\phi}[\chi_r(\tilde{x}_i)/r_i, \chi_e(\tilde{x}_i)/e_i]_{i=1}^n,\ \phi' \wedge \bigwedge_{i=1}^{n} \phi_i \not\Rightarrow false, \tilde{\phi}' \wedge \bigwedge_{i=1}^{n} \tilde{\phi}_i \not\Rightarrow false\}
\end{aligned}$$

The semantics of a variable $\tilde{x} \in \widetilde{\mathbb{V}}$ consists of two cases. If $\tilde{x}$ belongs to the environment, then the variable has been previously bound to a program expression S through a let-expression. In this case, the semantics of $\tilde{x}$ is exactly the semantics of S. If $\tilde{x}$ does not belong to the environment, then $\tilde{x}$ is a parameter of the function. Here, a new conditional error bound is added with two place holders, $\chi_r(\tilde{x})$ and $\chi_e(\tilde{x})$, representing the real value and the error of $\tilde{x}$, respectively.

The semantics of a floating-point arithmetic operation $\widetilde{op}$ is computed by composing the semantics of its operands. The real value is obtained by applying the correspondent real arithmetic operation op to the real values of the operands, and the new error bound is obtained by applying $\epsilon_{\widetilde{op}}$ to the errors and real values of the operands. The new conditions are obtained as the combination of

the conditions of the operands. Predicates ϕ_{op} and $\phi_{\widetilde{op}}$ represent the additional constraints needed when op and $\widetilde{op}$ are not total (as explained in Section 2).

The semantics of the expression *let* $\tilde{x} = \widetilde{A}$ *in* S updates the current environment by associating to variable $\tilde{x}$ the semantics of expression $\widetilde{A}$.

The semantics of the conditional uses an auxiliary operator $\Downarrow$ for propagating new information in the conditions.

Definition 5 (Condition propagation operator). *Given* $b \in \mathbb{B}$ *and* $\tilde{b} \in \widetilde{\mathbb{B}}$, $\langle \eta \rangle_t \twoheadrightarrow (r,e)^\pi \Downarrow_{(b,\tilde{b})} = \langle \bigcup_{(\phi,\tilde{\phi})\in\eta} (\phi \wedge b, \tilde{\phi} \wedge \tilde{b}) \rangle_t \twoheadrightarrow (r,e)^\pi$ *if* $\bigvee_{(\phi,\tilde{\phi})\in\eta} (\phi \wedge b \wedge \tilde{\phi} \wedge \tilde{b}) \not\Rightarrow$ *false, otherwise it is undefined. The definition of* $\Downarrow$ *naturally extends to sets of conditional error bounds: given* $C \in \mathbb{C}$, $C \Downarrow_{(b,\tilde{b})} = \bigcup_{c \in C} c \Downarrow_{(b,\tilde{b})}$.

The semantics of S_1 and S_2 are enriched with the information about the fact that real and floating-point execution paths match, i.e., both $\widetilde{B}$ and $R_{\mathbb{B}}(\widetilde{B})$ have the same value. If real and floating point execution paths do not coincide, the error of taking one branch instead of the other has to be considered. For example, if $\widetilde{B}$ is satisfied but $R_{\mathbb{B}}(\widetilde{B})$ is not, the *then* branch is taken in the floating point computation, but the *else* would have been taken in the real one. In this case, the error is the difference between the real value of the result of S_2 and the floating point result of S_1. It has been shown that this error is bounded by the round-off error of S_1 plus the difference between the real values of S_1 and S_2. The condition $(\neg R_{\mathbb{B}}(\widetilde{B}), \widetilde{B})$ is propagated in order to model that $\widetilde{B}$ holds but $R_{\mathbb{B}}(\widetilde{B})$ does not. The conditional error bounds representing this case are marked with **u**, denoting that the error is due to an unstable test. The parameter π of the semantics is augmented by one index that indicates the decision taken: 1 for the *then* and 0 for the *else* branch.

The semantics of a function call combines the conditions coming from the interpretation of the function and the ones coming from the semantics of the parameters. Variables representing real values and errors of formal parameters are replaced with the expressions coming from the semantics of the actual parameters.

The semantics of a program is a function $\mathcal{F} : \mathbb{P} \times Env \to \mathbb{C}$ defined as the least fixed point of the immediate consequence operator $\mathcal{P} : \mathbb{P} \times Env \times \mathbb{I} \to \mathbb{C}$, i.e., given $P \in \mathbb{P}$, $\mathcal{F}[\![P]\!] := lfp(\mathcal{P}[\![P]\!]_{\bot_{\mathbb{I}}})$, which is defined as follows for each function symbol f defined in P:

$$\mathcal{P}[\![P]\!]_I(f(\tilde{x}_1 \ldots \tilde{x}_n)) := \mathcal{E}[\![S]\!]^{\varepsilon}_{(\bot_{Env}, I)} \text{ if } f(\tilde{x}_1 \ldots \tilde{x}_n) = S \in P. \tag{3.2}$$

The least fixed point of $\mathcal{P}$ is guaranteed to exist from the Knaster-Tarski Fixpoint theorem [11] since $\mathcal{P}$ is monotonic over $\mathbb{C}$.

Example 2. Let P be a program composed by the declaration $f(\tilde{x}, \tilde{y}) =$ *if* $\tilde{x} > 1$ *then* 3 *elseif* $\tilde{y} \leq 2$ *then* $\tilde{x} \tilde{+} \tilde{y}$ *else* $\tilde{x} \widetilde{/} \tilde{y}$. The semantics of P is defined as $\mathcal{F}[\![P]\!] = \bigcup_{i=1}^{3} \{s_i\} \cup \bigcup_{i=1}^{6} \{u_i\}$ where the conditional error bounds s_i corresponding to the stable cases are:

$$s_1 = \langle R_{\mathbb{B}}(\tilde{x} > 1), \tilde{x} > 1 \rangle_{\mathbf{s}} \twoheadrightarrow (R(3), 0)^1$$

$$s_2 = \langle R_{\mathbb{B}}(\neg(\tilde{x} > 1)) \wedge R_{\mathbb{B}}(\tilde{y} \leq 2), \neg(\tilde{x} > 1) \wedge \tilde{y} \leq 2\rangle_{\mathbf{s}} \twoheadrightarrow (\chi_r(\tilde{x}) + \chi_r(\tilde{y}), \epsilon_{\tilde{+}}(\chi_r(\tilde{x}), \chi_e(\tilde{x}), \chi_r(\tilde{y}), \chi_e(\tilde{y})))^{01}$$

$$s_3 = \langle R_{\mathbb{B}}(\neg(\tilde{x} > 1)) \wedge R_{\mathbb{B}}(\neg(\tilde{y} \leq 2)) \wedge \chi_r(\tilde{x} \neq 0), \neg(\tilde{x} > 1) \wedge \neg(\tilde{y} \leq 2) \wedge \tilde{x} \neq 0\rangle_{\mathbf{s}} \twoheadrightarrow (\chi_r(\tilde{x})/\chi_r(\tilde{y}), \epsilon_{\tilde{/}}(\chi_r(\tilde{x}), \chi_e(\tilde{x}), \chi_r(\tilde{y}), \chi_e(\tilde{y})))^{00}$$

The conditional error bounds modeling unstable cases u_i are six and represent all the cases when real and floating-point flows diverge. For instance: $u_1 = \langle R_{\mathbb{B}}(\tilde{x} > 1), \neg(\tilde{x} > 1) \wedge \tilde{y} \leq 2\rangle_{\mathbf{u}} \twoheadrightarrow (R(3), |R(3) - (\chi_r(\tilde{x}) + \chi_r(\tilde{y}))| + \epsilon_{\tilde{+}}(\chi_r(\tilde{x}), \chi_e(\tilde{x}), \chi_r(\tilde{y}), \chi_e(\tilde{y})))^{\varepsilon}$ models a case in which the outermost conditional is unstable, and $u_2 = \langle R_{\mathbb{B}}(\neg(\tilde{x} > 1)) \wedge R_{\mathbb{B}}(\tilde{y} \leq 2), \neg(\tilde{x} > 1) \wedge \neg(\tilde{y} \leq 2)\rangle_{\mathbf{u}} \twoheadrightarrow (\chi_r(\tilde{x}) + \chi_r(\tilde{y}), |\chi_r(\tilde{x}) + \chi_r(\tilde{y}) - (\chi_r(\tilde{x})/\chi_r(\tilde{y}))| + \epsilon_{\tilde{/}}(\chi_r(\tilde{x}), \chi_e(\tilde{x}), \chi_r(\tilde{y}), \chi_e(\tilde{y})))^{\varepsilon}$ models a similar case for the inner conditional.

4 Abstraction Scheme

The semantics presented in Section 3 is not computable since the least fixed point of the operator defined in Equation (3.2) does not converge in a finite number of steps for recursive programs. In addition, the sound treatment of unstable tests provokes an explosion of the number of semantic elements generated when several nested if-then-else occur in a function. To overcome these problems, this section presents an abstraction framework for the semantics of Section 3 that limits the combinatory explosion due to nested if-then-else expressions. A widening operator is also defined to ensure the convergence of the analysis of recursive programs. This abstraction framework yields a computable abstract semantics that is suitable for the definition of a parametric static analysis of floating-point round-off errors. The proposed abstract semantics is parametric with respect to two Galois insertions:

- $(\mathbb{E}, \leq) \underset{\alpha_{\mathbb{E}}}{\overset{\gamma_{\mathbb{E}}}{\leftrightarrows}} (\dot{\mathbb{E}}, \dot{\leq})$ between (concrete) error expressions and abstract error expression in the complete lattice $(\dot{\mathbb{E}}, \dot{\leq}, \dot{\oplus}, \dot{\otimes}, \top_{\dot{\mathbb{E}}}, \bot_{\dot{\mathbb{E}}})$, where $\dot{\leq}$ is the order relation, $\dot{\oplus}$ is the least upper bound (*lub*), $\dot{\otimes}$ is the greatest lower bound (*glb*), $\top_{\dot{\mathbb{E}}}$ is the top, and $\bot_{\dot{\mathbb{E}}}$ is the bottom of the domain.
- $(\wp(\mathbb{B} \times \widetilde{\mathbb{B}}), \hat{\Rightarrow}) \underset{\alpha_{\mathbb{B}}}{\overset{\gamma_{\mathbb{B}}}{\leftrightarrows}} (\dot{\mathbb{B}}, \dot{\Rightarrow})$ between (concrete) conditions and abstract condition in the complete lattice $(\dot{\mathbb{B}}, \dot{\Rightarrow}, \dot{\vee}, \dot{\wedge}, \top_{\dot{\mathbb{B}}}, \bot_{\dot{\mathbb{B}}})$, where $\dot{\Rightarrow}$ is the order relation, $\dot{\vee}$ is the *lub*, $\dot{\wedge}$ is the *glb*, $\top_{\dot{\mathbb{B}}}$ is the top, and $\bot_{\dot{\mathbb{B}}}$ is the bottom.

These Galois insertions have to satisfy the following properties: $\alpha_{\mathbb{E}}(0) = \bot_{\dot{\mathbb{E}}}$, $\alpha_{\mathbb{B}}(\{\mathit{false}, \mathit{false}\}) = \bot_{\dot{\mathbb{B}}}$, and $\alpha_{\mathbb{B}}(\eta_1 \mathbin{\hat{\wedge}} \eta_2) = \alpha_{\mathbb{B}}(\eta_1) \mathbin{\dot{\wedge}} \alpha_{\mathbb{B}}(\eta_2)$.

The abstract semantics collects approximated information and stores it in an *abstract conditional error bound.*

Definition 6 (Abstract Conditional Error Bound). *An* abstract conditional error bound *is defined as a tuple of the form* $\langle\dot{\eta}\rangle_t \twoheadrightarrow (R, \dot{e})^{\pi}$, *where* $\dot{\eta} \in \dot{\mathbb{B}}$, $R \in \wp(\mathbb{A})$, $\dot{e} \in \dot{\mathbb{E}}$, $\pi \in \mathit{Path}$, *and* $t \in \{\mathbf{s}, \mathbf{u}\}$. *An abstract conditional error bound is valid when* $\dot{\eta} \not\dot{\Rightarrow} \bot_{\dot{\mathbb{B}}}$.

Abstract conditional error bounds are ordered in the following way: $\langle\dot{\eta}_1\rangle_{t_1} \twoheadrightarrow (R_1, \dot{e}_1)^{\pi_1} \leq \langle\dot{\eta}_2\rangle_{t_2} \twoheadrightarrow (R_2, \dot{e}_2)^{\pi_2} \iff \dot{\eta}_1 \dot{\Rightarrow} \dot{\eta}_2,\ R_1 \subseteq R_2,\ \dot{e}_1 \dot{\leq} \dot{e}_2,\ t_1 = t_2$, and $\pi_2 \leq_{prefix} \pi_1$.

The merge (collapse) of two abstract error bounds is defined as follows.

Definition 7. *Let $\langle\dot{\eta}_1\rangle_{t_1} \twoheadrightarrow (R_1, \dot{e}_1)^{\pi_1}$ and $\langle\dot{\eta}_2\rangle_{t_2} \twoheadrightarrow (R_2, \dot{e}_2)^{\pi_2}$ be two abstract conditional error bounds. Their merge is defined as $\langle\dot{\eta}_1\rangle_{t_1} \twoheadrightarrow (R_1, \dot{e}_1)^{\pi_1} \odot \langle\dot{\eta}_2\rangle_{t_2} \twoheadrightarrow (R_2, \dot{e}_2)^{\pi_2} := \langle\dot{\eta}_1 \dot{\vee} \dot{\eta}_2\rangle_{t_1} \twoheadrightarrow (R_1 \cup R_2, \dot{e}_1 \dot{\oplus} \dot{e}_2)^{mcp(\pi_1,\pi_2)}$ if $t_1 = t_2$, otherwise it is undefined.*

In Definition 7, the expression $mcp(\dot{\Pi})$ denotes the maximum common prefix of a set of decision paths $\dot{\Pi}$. For example, $mcp(\{0 \cdot 1 \cdot 0 \cdot 1, 0 \cdot 1 \cdot 0 \cdot 0, 0 \cdot 1\}) = 0 \cdot 1$.

As already mentioned, the concrete semantics of Section 3 computes one conditional error bound for every possible combination of real and floating-point execution path. This guarantees a sound treatment of unstable tests, but four different conditional error bounds are produced for each if-then-else. As a consequence, computing the semantics can become costly for programs with nested if-then-else expressions since the number of computed semantics elements grows exponentially. To overcome this limitation, an abstraction function is introduced to approximate sets of (concrete) conditional error bounds into sets of abstract ones. The main idea behind this abstraction is that the semantics is precisely computed just for a finite set of decision paths of interests, which are given as an input of the analysis. The conditional error bounds that correspond to other decision paths are collapsed together. Since, in general, the errors associated to unstable cases are several order of magnitude bigger than the ones due to floating-point rounding, stable and unstable cases are collapsed separately. This way, the abstraction does not lose too much precision.

The semantics presented in Section 3 is able to compute the conditions under which an unstable test occurs and to bound the error due to the difference between what is actually computed in the floating-point execution and what should have been computed in the ideal execution on real numbers. In general, this difference is large and, most of the times, one is interested just in knowing if unstable tests can occur in a program and under which circumstances. For this reason, the proposed abstraction collapses the unstable conditional error bounds in a unique expression. Using this approach, the abstract semantics is still able to soundly deal with unstable tests and to provide a sound approximation of the conditions under which the instability occurs. It also avoids the burden of differentiating each possible combination of real and floating-point paths that leads to an unstable test.

Given $\dot{\Pi} \in \wp(Path)$, let $\dot{\mathbb{C}}_{\dot{\Pi}}$ be the domain composed of sets of abstract conditional error bounds $\dot{C}$ such that for all $\langle\dot{\eta}\rangle_t \twoheadrightarrow (R, \dot{e})^{\pi} \in \dot{C}$ the following properties hold.

1. If there exists $\pi' \in \dot{\Pi}$ such that $\pi' \leq_{prefix} \pi$ then the cardinality of R is 1.
2. If $t = \mathbf{s}$ and there is no element in $\dot{C}$ of the form $\langle\dot{\eta}'\rangle_{\mathbf{s}} \twoheadrightarrow (R', \dot{e}')^{\pi'}$ different from $\langle\dot{\eta}\rangle_t \twoheadrightarrow (R, \dot{e})^{\pi}$ such that for all $\pi'' \in \dot{\Pi}$, it holds that $\pi'' \nleq_{prefix} \pi'$.

3. If $t = \mathbf{u}$, then there is no another unstable element in $\dot{C}$ of the form $\langle \dot{\eta}' \rangle_{\mathbf{u}} \twoheadrightarrow (R', \dot{e}')^{\pi'}$ different from $\langle \dot{\eta} \rangle_t \twoheadrightarrow (R, \dot{e})^{\pi}$.

Sets of abstract conditional error bounds in $\dot{\mathbb{C}}_{\dot{\Pi}}$ are (partially) ordered as follows. For all $\dot{C}_1, \dot{C}_2 \in \dot{\mathbb{C}}_{\dot{\Pi}}$, $\dot{C}_1 \dot{\sqsubseteq} \dot{C}_2$ if and only if for all $\dot{c}_1 \in \dot{C}_1$ $\exists \dot{c}_2 \in \dot{C}_2$. $\dot{c}_1 \leq \dot{c}_2$. The equivalence relation derived from $\dot{\sqsubseteq}$ is defined as $\dot{C}_1 \dot{\equiv} \dot{C}_2$ if and only if $\dot{C}_1 \dot{\sqsubseteq} \dot{C}_2 \wedge \dot{C}_2 \dot{\sqsubseteq} \dot{C}_1$. In the following, by abuse of notation, the quotient of $\dot{\sqsubseteq}$ over equivalence classes will be denoted with the same symbol. Furthermore, sets of conditional error bounds will be used modulo $\dot{\equiv}$ and their class will be denoted as $\dot{\mathbb{C}}_{\dot{\Pi}}$. Given $\dot{C}_1, \dot{C}_2 \in \dot{\mathbb{C}}_{\dot{\Pi}}$, their least upper bound is defined as follows

$$
\begin{aligned}
\dot{C}_1 \dot{\sqcup} \dot{C}_2 := \; & [\bigcup \{ \langle \dot{\eta} \rangle_t \twoheadrightarrow (R, \dot{e})^{\pi} \in \dot{C}_1 \cup \dot{C}_2 \mid \exists \pi' \in \dot{\Pi}.\ \pi' \leq_{prefix} \pi,\ t = \mathbf{s} \}]_{\dot{\equiv}} \cup \\
& \bigodot \{ \langle \dot{\eta} \rangle_t \twoheadrightarrow (R, \dot{e})^{\pi} \in \dot{C}_1 \cup \dot{C}_2 \mid \nexists\, \pi' \in \dot{\Pi}.\ \pi' \leq_{prefix} \pi,\ t = \mathbf{s} \} \cup \\
& \bigodot \{ \langle \dot{\eta} \rangle_t \twoheadrightarrow (R, \dot{e})^{\pi} \in \dot{C}_1 \cup \dot{C}_2 \mid t = \mathbf{u} \}
\end{aligned}
\tag{4.1}
$$

The tuple $(\dot{\mathbb{C}}_{\dot{\Pi}}, \dot{\sqsubseteq}, \dot{\bigsqcup}, \dot{\bigsqcap}, \top_{\dot{\mathbb{C}}_{\dot{\Pi}}}, \varnothing)$ is a complete lattice, where $\top_{\dot{\mathbb{C}}_{\dot{\Pi}}} := \bigcup \{ \langle \top_{\dot{\mathbb{B}}} \rangle_t \twoheadrightarrow (\mathbb{R}, \top_{\dot{\mathbb{E}}})^{\varepsilon} \mid t \in \{\mathbf{u}, \mathbf{s}\} \}$ is the greatest element of $\dot{\mathbb{C}}_{\dot{\Pi}}$, $\varnothing$ is the least element, and the greatest lower bound ($\dot{\bigsqcap}$) is defined as follows $\dot{C}_1 \dot{\sqcap} \dot{C}_2 := [\{ \dot{c} \in \dot{\mathbf{C}} \mid \exists \dot{c}_1 \in \dot{C}_1 . \dot{c} \leq \dot{c}_1,\ \exists \dot{c}_2 \in \dot{C}_2 . \dot{c} \leq \dot{c}_2 \}]_{\dot{\equiv}}$.

Given $\dot{\Pi} \in \wp(Path)$, the abstraction function $\alpha_{\dot{\Pi}}$ collapses together all the stable abstract conditional error bounds that are not produced from a path in $\dot{\Pi}$. In addition, it collapses all the unstable conditional error bounds in a unique one. The abstraction function $\alpha_{\dot{\Pi}}$ and its adjoint $\gamma_{\dot{\Pi}}$ are defined as follows and form a Galois insertion $(\mathbb{C}, \sqsubseteq) \underset{\alpha_{\dot{\Pi}}}{\overset{\gamma_{\dot{\Pi}}}{\leftrightarrows}} (\dot{\mathbb{C}}_{\dot{\Pi}}, \dot{\sqsubseteq})$.

Definition 8. *Let $\dot{\Pi} \in \wp(Path)$, $C \in \mathbb{C}$ and $\dot{C} \in \dot{\mathbb{C}}_{\dot{\Pi}}$, the abstraction and concretization functions are defined as follows.*

$$
\begin{aligned}
\alpha_{\dot{\Pi}}(C) := \; & \dot{\bigsqcup} \{ \langle \alpha_{\mathbb{B}}(\eta) \rangle_t \twoheadrightarrow (\{r\}, \alpha_{\mathbb{E}}(e))^{\pi} \mid \langle \eta \rangle_t \twoheadrightarrow (r, e)^{\pi} \in C, \\
& \qquad \exists \pi' \in \dot{\Pi} . \pi' \leq_{prefix} \pi,\ t = \mathbf{s} \} \ \dot{\sqcup} \\
& \bigodot \{ \langle \alpha_{\mathbb{B}}(\eta) \rangle_t \twoheadrightarrow (\{r\}, \alpha_{\mathbb{E}}(e))^{\pi} \mid \langle \eta \rangle_t \twoheadrightarrow (r, e)^{\pi} \in C, \\
& \qquad \nexists\, \pi' \in \dot{\Pi} . \pi' \leq_{prefix} \pi,\ t = \mathbf{s} \} \ \dot{\sqcup} \\
& \bigodot \{ \langle \alpha_{\mathbb{B}}(\eta) \rangle_t \twoheadrightarrow (\{r\}, \alpha_{\mathbb{E}}(e))^{\pi} \mid \langle \eta \rangle_t \twoheadrightarrow (r, e)^{\pi} \in C,\ t = \mathbf{u} \} \\
\gamma_{\dot{\Pi}}(\dot{C}) := \; & \bigsqcup \{ \langle \gamma_{\mathbb{B}}(\dot{\eta}) \rangle_t \twoheadrightarrow (r, \gamma_{\mathbb{E}}(\dot{e}))^{\pi} \mid \exists \langle \dot{\eta} \rangle_t \twoheadrightarrow (R, \dot{e})^{\pi} \in \dot{C},\ r \in R \}
\end{aligned}
$$

Lemma 1. *Given $\dot{\Pi} \in \wp(Path)$, the pair of functions $(\alpha_{\dot{\Pi}}, \gamma_{\dot{\Pi}})$ is a Galois insertion between $(\mathbb{C}, \sqsubseteq)$ and $(\dot{\mathbb{C}}_{\dot{\Pi}}, \dot{\sqsubseteq})$.*

Given $\dot{\Pi} \in \wp(Path)$, an *abstract environment* is defined as a function mapping a variable to a set of abstract conditional error bounds, i.e., $\dot{Env}_{\dot{\Pi}} = \widetilde{\mathbb{V}} \to \dot{\mathbb{C}}_{\dot{\Pi}}$. The empty abstract environment is denoted as $\bot_{\dot{Env}}$ and maps every variable to the empty set $\varnothing$.

Given $\bar{\Pi} \in [\mathbb{M} \to \wp(Path)]$, an *abstract interpretation* is a function $\dot{I}$ such that $\forall f(\tilde{x}_i)_{i=1}^n \in \mathbb{M}$, $\dot{I}(f(\tilde{x}_i)_{i=1}^n) \in \dot{\mathbb{C}}_{\bar{\Pi}(f(\tilde{x}_i)_{i=1}^n)}$ modulo variance. The set of all interpretations respecting the aforementioned property is denoted as $\dot{\mathbb{I}}_{\bar{\Pi}}$. The empty interpretation is denoted as $\bot_{\dot{\mathbb{I}}_{\bar{\Pi}}}$ and maps everything to the empty set. The Galois insertion of Definition 8 can be lifted to the interpretation level in the following way.

Definition 9. *Let $\bar{\Pi} \in [\mathbb{M} \to \wp(Path)]$, given $I \in \mathbb{I}$ and $\dot{I} \in \dot{\mathbb{I}}_{\bar{\Pi}}$, the abstraction function for interpretations and its adjoint are defined as follows for every function $f(\tilde{x})_{i=1}^n$ defined in I.*

$$\bar{\alpha}_{\bar{\Pi}}(I)(f(\tilde{x})_{i=1}^n) \coloneqq \alpha_{\bar{\Pi}(f(\tilde{x})_{i=1}^n)}(I(f(\tilde{x})_{i=1}^n))$$
$$\bar{\gamma}_{\bar{\Pi}}(\dot{I})(f(\tilde{x})_{i=1}^n) \coloneqq \gamma_{\bar{\Pi}(f(\tilde{x})_{i=1}^n)}(\dot{I}(f(\tilde{x})_{i=1}^n))$$

Lemma 2. *Given $\bar{\Pi} \in [\mathbb{M} \to \wp(Path)]$, $(\bar{\alpha}_{\bar{\Pi}}, \bar{\gamma}_{\bar{\Pi}})$ is a Galois insertion between $(\mathbb{I}, \sqsubseteq)$ and $(\dot{\mathbb{I}}_{\bar{\Pi}}, \dot{\sqsubseteq})$, where $\sqsubseteq$ and $\dot{\sqsubseteq}$ denotes the natural extension of these order relations to interpretations.*

Given $\bar{\Pi} \in [\mathbb{M} \to \wp(Path)]$, abstract interpretation theory [12] defines the best correct abstract version of the semantic operator $\mathcal{P}$ with respect to the Galois insertion $(\alpha_{\dot{\Pi}}, \gamma_{\dot{\Pi}})$ simply as the composition $\alpha_{\dot{\Pi}} \circ \mathcal{P} \circ \gamma_{\dot{\Pi}}$. Abstract interpretation theory [12] ensures that the abstract fixpoint semantics $\dot{\mathcal{F}}^{\dot{\Pi}} \coloneqq lfp(\dot{\mathcal{P}}^{\dot{\Pi}})$ is the best correct approximation of $\mathcal{F}$. It is correct because $\alpha_{\dot{\Pi}}(\mathcal{F}) \dot{\sqsubseteq} \dot{\mathcal{F}}^{\dot{\Pi}}$ and it is the best because it is the minimum (with respect to $\dot{\sqsubseteq}$) of all correct approximations.

Example 3. Consider the program of Example 2 and its concrete semantics. Suppose that the selected decision path of interest is 01 and the error expressions and conditions abstraction functions are the identity. The abstract semantics of P is defined as $\dot{\mathcal{F}}^{\{01\}}[\![P]\!] = s_2 \,\dot{\sqcup}\, (s_1 \odot s_3) \,\dot{\sqcup}\, \bigodot_{i=1}^{6} u_i$.

The conditional error bound s_2, corresponding to the decision path of interest 01, is computed precisely. The other two stable bounds are collapsed together in one abstract conditional error bound of the form $s_1 \odot s_3 = \langle R_{\mathbb{B}}(\tilde{x} > 1) \vee (R_{\mathbb{B}}(\neg(\tilde{x} > 1)) \wedge R_{\mathbb{B}}(\neg(\tilde{y} \le 2)) \wedge \chi_r(\tilde{x} \ne 0)), \tilde{x} > 1 \vee (\neg(\tilde{x} > 1) \wedge \neg(\tilde{y} \le 2) \wedge \tilde{x} \ne 0)\rangle_{\mathbf{s}} \twoheadrightarrow (\{R(3), \chi_r(\tilde{x})/\chi_r(\tilde{y})\}, \epsilon_{\bar{7}}(\chi_r(\tilde{x}), \chi_e(\tilde{x}), \chi_r(\tilde{y}), \chi_e(\tilde{y})))^{\varepsilon}$. The unstable cases are collapsed together in $\bigodot_{i=1}^{6} u_i$.

Widening operators [12, 13] provide a solution to the convergence problem by over-approximating infinite increasing chains in a finite number of steps. A widening operator for the domain of abstract conditional error bounds is defined. Intuitively, it approximates to the top of the domain when the recursion is possibly non terminating (the conditions are not changing and the error is growing), otherwise it tries to converge in k steps for recursion calls that could terminate (the conditions are changing and they are converging to *false*).

Definition 10. *Given $\bar{\Pi} \in [\mathbb{M} \to \wp(Path)]$, $A_1, A_2 \in \dot{\mathbb{C}}_{\dot{\Pi}}$ such that $\dot{C}_1 \dot{\sqsubseteq} \dot{C}_2$, $n_1, n_2 \in \mathbb{N}$ such that $n_1 \leq n_2$, and $k \in \mathbb{N}$, the operator $\triangledown_k : (\dot{\mathbb{C}}_{\dot{\Pi}} \times \mathbb{N}) \times (\dot{\mathbb{C}}_{\dot{\Pi}} \times \mathbb{N}) \to (\dot{\mathbb{C}}_{\dot{\Pi}} \times \mathbb{N})$ is defined as follows.*

$$
\begin{aligned}
&(\dot{C}_1, n_1) \triangledown_k (\dot{C}_2, n_2) := \\
&\quad \Big(\dot{\bigsqcup}\{\langle \dot{\eta}_2 \rangle_{t_2} \twoheadrightarrow (R_2, \top_{\dot{\mathbb{E}}})^{\pi_2} \in \dot{C}_2 \mid \langle \dot{\eta}_2 \rangle_{t_2} \twoheadrightarrow (R_2, \dot{e}_2)^{\pi_2} \in \dot{C}_2, (n_2 > k \textit{ or} \\
&\quad (\exists \langle \dot{\eta}_1 \rangle_{t_1} \twoheadrightarrow (R_1, \dot{e}_1)^{\pi_1} \in \dot{C}_1 \textit{ such that } \dot{\eta}_1 \leftrightarrow \dot{\eta}_2,\ R_1 \subseteq R_2 \textit{ and } \dot{e}_1 \dot{<} \dot{e}_2)\ \}\ \dot{\sqcup} \\
&\quad \dot{\bigsqcup}\{\langle \dot{\eta}_2 \rangle_{t_2} \twoheadrightarrow (R_2, \dot{e}_2)^{\pi_2} \in \dot{C}_2 \mid \langle \dot{\eta}_2 \rangle_{t_2} \twoheadrightarrow (R_2, \dot{e}_2)^{\pi_2} \in \dot{C}_2, n_2 \leq k, \\
&\quad (\nexists\ \langle \dot{\eta}_1 \rangle_{t_1} \twoheadrightarrow (R_1, \dot{e}_1)^{\pi_1} \in \dot{C}_1 \textit{ such that } \dot{\eta}_1 \leftrightarrow \dot{\eta}_2,\ R_1 \subseteq R_2 \textit{ and } \dot{e}_1 \dot{<} \dot{e}_2)\ \}, n_2\Big)
\end{aligned}
$$

Lemma 3. *Given $k \in \mathbb{N}$ and $\bar{\Pi} \in [\mathbb{M} \to \wp(Path)]$, the operator $\triangledown_k$ is a widening operator on $(\dot{\mathbb{C}}_{\dot{\Pi}} \times \mathbb{N})$.*

Because of Lemma 3 and the results in [12, 13] it is guaranteed that, for any $k \in \mathbb{N}$, $\bar{\Pi} \in [\mathbb{M} \to \wp(Path)]$, program $P \in \mathbb{P}$ and function $f(\tilde{x})_{i=1}^n$ defined in P, the chain defined as follows converges in a finite number of steps.

$$
\begin{aligned}
&(\dot{I}_0(f(\tilde{x})_{i=1}^n), n_0) = (\varnothing, 0) \\
&(\dot{I}_{i+1}(f(\tilde{x})_{i=1}^n), n_{i+1}) = \begin{cases} (\dot{I}_i(f(\tilde{x})_{i=1}^n), n_i) & \text{if } \dot{\mathcal{P}}^{\bar{\Pi}}[\![P]\!]_{\dot{I}_i}(f(\tilde{x})_{i=1}^n) \dot{\sqsubseteq} I_i(f(\tilde{x})_{i=1}^n) \\ & \text{and } n_i \leq n_{i+1} \\ (\dot{I}_i(f(\tilde{x})_{i=1}^n), n_i) \triangledown_k (\dot{\mathcal{P}}^{\bar{\Pi}}[\![P]\!]_{\dot{I}_i}(f(\tilde{x})_{i=1}^n), n_i + 1) \\ & \text{otherwise} \end{cases}
\end{aligned}
$$

5 PRECiSA

This section presents the prototype tool PRECiSA[4] (Program Round-off Error Certifier via Static Analysis) that implements a possible instantiation of the abstraction framework defined in Section 4. This tool is an enhancement of the tool presented in [6]. PRECiSA supports the basic arithmetic operations (addition, subtraction, multiplication, and division), square root, logarithm, exponential, trigonometric functions, floor, and absolute value. As illustrated in Fig. 1, PRECiSA accepts as inputs a program written in a simple functional language that follows the grammar in Section 4 or in PVS syntax, initial ranges for the input variables of the program, and a set of computational paths of interest for each function in the input program.

PRECiSA computes the abstract semantics presented in Section 4. The conditional error bounds corresponding to the execution paths selected by the user are computed precisely, while the others are collapsed together. A decision path of interest intuitively corresponds to a subprogram or subexpression inside a function of the input program. If the user does not select any subprogram of interest, the tool will just produce the overall round-off error for the stable case and for the unstable one.

[4] The web-interface of PRECiSA is available at `http://precisa.nianet.org`.

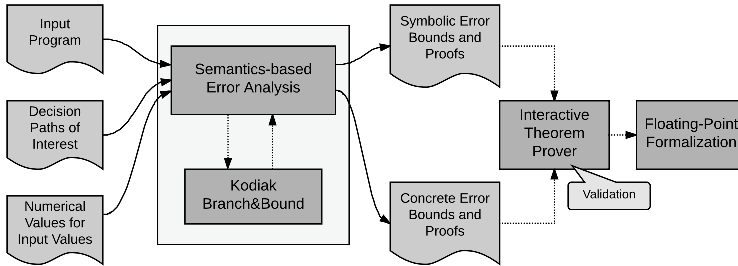


Fig. 1. Functional architecture of PRECiSA.

No additional abstraction is done for errors and conditions. Thus, the Galois insertions are simply defined as $(\mathbb{E}) \overset{id}{\underset{id}{}} (\mathbb{E})$ and $(\mathbb{B} \;\; \mathbb{B}),) \overset{id}{\underset{id}{}} (\mathbb{B} \;\; \mathbb{B}),)$. The merge of abstract conditional error bounds will be instantiated as follows $\dot{\eta}_1 \; t_1 \;\; (\;\; R_1, \dot{e}_1)^{\pi_1} \;\; \dot{\eta}_2 \; t_2 \;\; (\;\; R_2, \dot{e}_2)^{\pi_2} = \;\; \dot{\eta}_1 \; \hat{} \; \dot{\eta}_2 \; t_1 \; (R_1 \;\; R_2, max(\dot{e}_1, \dot{e}_2))^{mcp(\pi_1,\pi_2)}$.

The semantics presented in Section 4 is completely independent from the input values provided to the program. This makes the proposed approach scalable since it enables a compositional analysis that reuses already computed results. However, given the initial ranges for the input variables, it is essential to compute numerical bounds from the (abstract) conditional error bounds. To this aim, the proposed prototype tool uses the optimizer Kodiak [14] which is based on the formally verified branch-and-bound algorithm presented in [15]. This branch-and-bound algorithm relies on enclosure functions for arithmetic operators. These enclosure functions compute provably correct over approximations of the symbolic error expressions using either interval arithmetic or Bernstein basis. The algorithm recursively splits the domain of the function into smaller subdomains and computes an enclosure of the original expression in these subdomains. The recursion stops when a precise enclosure is found, based on a given precision, or when a given maximum recursion depth is reached. The output of the algorithm is a numerical enclosure for each symbolic error expression.

Besides computing error bounds, PRECiSA generates proof certificates ensuring that these bounds are correct. Having an externally checkable certificate increases the level of trustworthiness of the proposed tool. PRECiSA relies on the higher-order logic interactive theorem prover PVS [16] and a floating-point formalization originally presented in [9] and extended in [6]. Therefore, each computed conditional error bound is translated into a lemma stating that, provided the conditions are satisfied, the floating-point value resulting from the execution of f on floating-point values differs from the exact real-number computation by at most the round-off error approximation computed by the semantics. PRECiSA generates proof scripts that automatically discharge the generated lemmas.

In the following, PRECiSA is compared in terms of accuracy and performance with the following floating-point analysis tools: Gappa (ver. 1.3.1) [5],

Fluctuat (ver. 3.1376) [4], FPTaylor (ver. 0.9) [2], Real2Float [17], and Rosa [3] (see Section 6 for a description of each tool). This comparison was performed using benchmarks taken from the Rosa and FPTaylor repositories. The selected benchmarks involve nonlinear expressions and polynomial approximations of functions, taken from equations used in physics, control theory, and biological modeling. In addition, some extra benchmarks taken from real-world avionics algorithms are considered. The experimental environment consisted of a 2.5 GHz Intel Core i7-4710MQ with 24 GB of RAM, running under Ubuntu 16.04 LTS. The benchmarks presented in this section and the corresponding proof certificates are available as part of the PRECiSA distribution.[5]

Table 1 shows numerical round-off error bounds computed by the aforementioned tools. Since the considered tools offers different configurations and options for the analysis, only the best estimation obtained by each tool for each example is reported in the table. In fact, FPTaylor offers two different optimization algorithms and two different rounding models. Gappa and Fluctuat allow the user to manually provide hints to obtain tighter error bounds. For the sake of uniformity, for all examples and tools, input variables and constants are assumed to be real numbers. This means that they carry a round-off error that has to be taken into consideration in the analysis. PRECiSA compares favorably to the other tools in terms of precision. Additionally, it supports a large set of basic and transcendental operators as well as common programming languages constructs such as conditionals and loops. On the contrary, some of the other tools lack that support, hence, they cannot analyze all the benchmarks. For instance, the floor operator appears in the cpr_yz0 and it is not supported by Real2Float, Rosa, and FPTaylor. Stynlinski and PolyCARP contain conditionals that are not handled by FPTaylor and Gappa. PRECiSA is the only tool that is able to analyze the recursive program mult_pow2_rec.

The times for the computation of the bounds in Table 1 are shown in Table 2. Overall, Fluctuat is the fastest approach but it does not produce certificates for the soundness of its results. The performance of PRECiSA is in line with similar tools for most of the examples, and for some of the considered benchmarks PRECiSA is the fastest approach.

In summary, for the considered examples, the proposed tool provides a good trade-off between accuracy and performance together with a wide support for arithmetic operations and programming constructs.

6 Related Work

The use of abstract interpretation and semantics based approaches for the problem of analyzing floating-point programs is not new. The static analyzer Astrée [18] automatically detects the presence of potential floating-point run-time exceptions such as overflows by means of sound floating-point abstract domains [19,20]. The abstraction scheme presented here shares some similarities with the ap-

[5] The PRECiSA distribution is available at `https://github.com/nasa/PRECiSA`.

	Gappa	Fluctuat	Real2Float	Rosa	FPTaylor	PRECiSA
azimuth	n/a	n/a	2.83E-13	n/a	**8.32E-15**	*1.19E-13*
carbonGas	*6.01E-09*	1.17E-08	2.21E-08	1.60E-08	**5.90E-09**	7.17E-09
doppler1	1.61E-13	*1.27E-13*	7.65E-12	2.68E-13	**1.22E-13**	1.98E-13
doppler2	2.86E-13	*2.35E-13*	1.57E-11	6.45E-13	**2.23E-13**	3.81E-13
doppler3	8.69E-14	*7.12E-14*	8.59E-12	1.01E-13	**6.63E-14**	1.09E-13
himmilbeau	**8.51E-13**	*1.00E-12*	1.42E-12	*1.00E-12*	*1.00E-12*	*1.00E-12*
jet	4.45E+03	1.07E-10	n/a	4.91E-09	**1.03E-11**	*1.59E-11*
kepler0	1.09E-13	1.03E-13	1.20E-13	*8.28E-14*	**7.47E-14**	1.06E-13
kepler1	4.68E-13	*3.51E-13*	4.67E-13	4.14E-13	**2.86E-13**	3.90E-13
kepler2	2.38E-12	2.24E-12	2.09E-12	2.15E-12	*1.58E-12*	**1.53E-12**
predatorPrey	*1.67E-16*	2.35E-16	2.51E-16	1.98E-16	**1.59E-16**	1.84E-16
rigidBody1	**2.95E-13**	*3.22E-13*	5.33E-13	*3.22E-13*	**2.95E-13**	**2.95E-13**
rigidBody2	*3.61E-11*	3.65E-11	6.48E-11	3.65E-11	*3.61E-11*	**3.60E-11**
sine	6.91E-16	7.41E-16	6.03E-16	*5.18E-16*	**3.87E-16**	6.37E-16
sineOrder3	*6.54E-16*	1.09E-15	1.19E-15	9.96E-16	**5.94E-16**	1.17E-15
sphere	n/a	n/a	1.52E-14	n/a	**8.11E-15**	*9.99E-15*
sqroot	5.35E-16	6.83E-16	1.28E-15	6.18E-16	*5.01E-16*	**4.29E-16**
t_div_t1	9.99E+00	2.80E-12	*8.53E-16*	5.68E-11	**2.22E-16**	3.91E-15
turbine1	2.41E-14	3.09E-14	2.46E-11	5.99E-14	**1.66E-14**	*2.17E-14*
turbine2	3.32E-14	*2.59E-14*	2.07E-12	7.67E-14	**1.99E-14**	2.81E-14
turbine3	3.52E-01	1.34E-14	1.70E-11	4.62E-14	**9.55E-15**	*1.22E-14*
verhulst	*2.84E-16*	4.80E-16	4.66E-16	4.67E-16	**2.47E-16**	3.74E-16
PolyCARP (stable)	n/a	*1.89E-15*	n/a	n/a	n/a	**1.83E-15**
PolyCARP (unstable)	n/a	n/a	6.60E+00	n/a	n/a	**6.00E-01**
Stynlinski (stable)	n/a	**2.29E-14**	n/a	*2.31E-14*	n/a	4.28E-14
Stynlinski (unstable)	n/a	**2.29E-14**	n/a	*2.31E-14*	n/a	1.61E+02
cpr_yz0	*1.35E+05*	**1.31E+05**	n/a	n/a	n/a	**1.31E+05**
logExp	n/a	n/a	*2.52E-15*	n/a	**1.49E-15**	3.22E-15
hartman3	n/a	n/a	2.99E-13	n/a	**3.26E-15**	*1.58E-14*
hartman6	n/a	n/a	5.07E-13	n/a	**5.26E-15**	*2.24E-13*
mult_pow2_rec (stable)	n/a	n/a	n/a	n/a	n/a	**7.11E-15**

Table 1. Experimental results for absolute round-off error bounds (**bold** indicates the best approximation, *italic* indicates the second best.)

proach of [21] where the analysis is refined by partitioning the program with respect to its control flow.

Some semantics-based approaches have been proposed to estimate the round-off error of a program. In [22], a family of abstract semantics parametric with respect to the error order and to a partition of the program is proposed for floating-point round-off errors. In [23], several abstract semantics for the static analysis of finite precision computations are defined. In contrast to the approach presented in this paper, the abstract semantics in [22] and [23] are not compositional since in these approaches the error is computed starting from a set of input ranges for the initial variables.

Diverse analysis techniques and tools to estimate the round-off error of floating-point computations have been proposed in the literature. Fluctuat [4] is a commercial analyzer that accepts as input a C program with annotations about input bound and uncertainties, and it produces bounds for the round-off error of the program expressions decomposed with respect to its provenance. Fluctuat uses a zonotopic abstract domain [23] that is based on affine arithmetic [24]. It is able to soundly treat unstable tests as explained in [25] and it provides support for iterative programs by using the widening operators introduced in [26, 27]. The widening operator presented in this paper is different from the ones of [26, 27] in that it takes advantage of the information contained in the path conditions.

	Gappa	Fluctuat	Real2Float	Rosa	FPTaylor	PRECiSA
azimuth	n/a	n/a	**1.986**	n/a	26.050	*5.204*
carbonGas	2.130	**0.062**	0.776	26.734	0.497	*0.090*
doppler1	3.475	6.904	5.957	17.293	*1.280*	**0.447**
doppler2	3.456	6.835	5.934	18.336	*1.504*	**0.402**
doppler3	3.604	6.837	5.846	26.996	*1.449*	**0.419**
himmilbeau	1.636	**0.013**	0.193	4.478	0.473	*0.106*
jet	8.604	**1.033**	n/a	264.811	*2.457*	255.278
kepler0	8.107	9.467	**0.203**	3.377	*2.813*	2.878
kepler1	2.088	**0.430**	8.509	132.313	*1.642*	8.964
kepler2	9.303	*2.233*	6.630	63.256	**0.743**	345.785
predatorPrey	1.259	**0.019**	0.684	26.452	0.521	*0.021*
rigidBody1	*0.030*	**0.013**	0.434	0.298	0.427	0.049
rigidBody2	*0.047*	**0.014**	0.272	2.752	0.470	1.035
sine	4.147	**0.022**	0.872	4.513	*0.625*	0.631
sineOrder3	1.966	**0.017**	0.296	0.771	0.437	*0.021*
sphere	n/a	n/a	*0.033*	n/a	35.116	**0.020**
sqroot	4.968	**0.014**	0.713	1.328	43.428	*0.047*
t_div_t1	0.160	**0.017**	34.656	6.207	0.418	*0.021*
turbine1	6.222	*5.410*	67.599	19.254	62.760	**1.746**
turbine2	4.185	4.311	*3.927*	6.483	44.138	**2.003**
turbine3	6.927	*5.417*	66.991	20.642	62.623	**4.569**
verhulst	0.346	**0.018**	0.425	7.730	0.418	*0.019*
Polycarp (stable)	n/a	**0.013**	n/a	n/a	n/a	*0.018*
Polycarp (unstable)	n/a	n/a	*0.024*	n/a	n/a	**0.018**
Stynlinski (stable)	n/a	**0.266**	n/a	58.543	n/a	*16.376*
Stynlinski (unstable)	n/a	**0.313**	n/a	58.543	n/a	*16.376*
yz0	7.177	**0.014**	n/a	n/a	n/a	*0.249*
logExp	n/a	n/a	0.664	n/a	*0.389*	**0.026**
hartman3	n/a	n/a	**1.760**	n/a	84.147	*44.309*
hartman6	n/a	n/a	**87.582**	n/a	*2191.622*	4320.212
mult_pow2_rec (stable)	n/a	n/a	n/a	n/a	n/a	**0.037**

Table 2. Times in seconds for the generation of round-off error bounds and certificates (**bold** indicates the best time, *italic* indicates the second best.)

RangeLab [28] is an interactive tool that determines the range of the round-off errors for elementary arithmetic expression based on the semantics of [22]. RangeLab is able to deal with while loops by means of a widening operator based on the classical interval domain widening. However, it does not provides a sound approximation of unstable conditionals. RangeLab and Fluctuat do not generate formal certificates for the computed bounds and they are not compositional.

FPTaylor [2] uses symbolic Taylor expansions to approximate floating-point expressions and applies a global optimization technique to obtain tight bounds for round-off errors. In addition, FPTaylor emits certificates for HOL Light [29] except for the configurations that use an improved rounding model that correlates error terms and allows much tighter error bounds [7]. Because of the technique used by FPTaylor, it is restricted to smooth functions. Unlike PRECiSA, which targets programs with conditional and function calls, FPTaylor is designed to analyze arithmetic expressions. FPTuner [1] uses FPTaylor to implement a rigorous approach to precision allocation of mixed-precision arithmetic expressions.

VCFloat [30] is a tool that automatically computes round-off error terms for numerical C expressions along with their correctness proof in Coq. This tool uses interval arithmetic to approximate the error bounds and generates validity conditions on the expressions. Similarly to FPTaylor, VCFloat targets only

arithmetic expressions. Real2Float [17] computes certified bounds for round-off errors by using an optimization technique employing semidefinite programming and sum of square certificates. Real2Float at the moment does not handle denormal floating-point numbers nor loops. Gappa [5] computes enclosures for floating-point expressions via interval arithmetic. This enclosure method enables a quick computation of the bounds, but may result in pessimistic error estimations. Gappa also generates a proof of the results that can be checked in the Coq proof assistant. In Gappa, the bound computation, the certification construction, and their verification may require hints from the user. Thus, some level of expertise is required, unlike PRECiSA, which is fully automatic. Rosa [3,31] uses a compilation algorithm that, from an ideal real-valued implementation, produces a finite-precision version (if it exists) that is guaranteed to meet a given desired precision. Rosa soundly deals with unstable tests and with bounded loops with bounded variables.

7 Conclusion

In this paper, a semantic framework based on abstract interpretation has been presented with the aim of providing a parametric round-off error static analysis for floating-point programs. The abstract semantics defined by this framework enjoys several features. It is defined in a compositional way, which allows for an incremental, modular, and efficient treatment of the program being analyzed. This makes the analysis defined upon this framework scalable and reusable. Moreover, the semantics is able to deal with any floating-point operator provided the existence of a round-off error estimation that satisfies Formula (2.5). Finally, recursion and conditionals are soundly handled.

The semantic analysis proposed in this paper is sound with respect to unstable tests and it associates conditions to the computed error estimation. This makes the analysis more precise since different execution paths may lead to different round-off errors. The proposed technique also avoids considering computations that lead to runtime errors such as division by zero or square root of a negative number. Additionally, the information collected in the conditions is used to discard impossible execution paths and to characterize initial input values that may cause large round-off errors.

PRECiSA is an implementation of the proposed framework that, additionally, generates proof certificates ensuring the correctness of the computed error bounds. In future work, the authors plan to integrate in PRECiSA other abstract domains such as affine arithmetic and a compositional version of the symbolic Taylor expansions of [2]. This way, the most suitable domain can be chosen depending on the input program and on the desired tradeoff between efficiency and precision. Another interesting future direction is the integration of PRECiSA with the static analyzer Frama-C [32]. This integration will enable the automated formal verification of C floating-point programs.

References

1. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of POPL 2017, ACM (2017) 300–315
2. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In: Proceedings of FM 2015. Volume 9109 of Lecture Notes in Computer Science., Springer (2015) 532–550
3. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of POPL 2014, ACM (2014) 235–248
4. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Proceedings of SAS 2006. Volume 4134 of Lecture Notes in Computer Science., Springer (2006) 18–34
5. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. on Computers **60**(2) (2011) 242–253
6. Moscato, M.M., Titolo, L., Dutle, A., Muñoz, C.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Proceedings of the International Conference on Computer Safety, Reliablilty, and Security, SAFECOMP 2017. (2017)
7. Baranowski, M., Briggs, I., Chiang, W., Gopalakrishnan, G., Rakamaric, Z., Solovyev, A.: Moving the Needle on Rigorous Floting-point Precision Tuning. 6th Workshop on Automated Formal Methods (AFM 2017) (2017)
8. Daumas, M., Rideau, L., Théry, L.: A generic library for floating-point numbers and its application to exact computing. In: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics", Springer Berlin Heidelberg (2001) 169–184
9. Boldo, S., Muñoz, C.: A high-level formalization of floating-point numbers in PVS. Technical Report CR-2006-214298, NASA (2006)
10. Goldberg, D.: What Every Computer Scientist Should Know About Floating-point Arithmetic. ACM Comput. Surv. **23**(1) (1991) 5–48
11. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. In: Pacific Journal of Mathematics. (1955) 285–309
12. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of POPL 1977, ACM (1977) 238–252
13. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise Widening Operators for Convex Polyhedra. Science of Computer Programming **58**(1-2) (2005) 28–56
14. Smith, A.P., Muñoz, C.A., Narkawicz, A.J., Markevicius, M.: A rigorous generic branch and bound solver for nonlinear problems. In: 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2015, Timisoara, Romania, September 21-24, 2015. (2015) 71–78
15. Narkawicz, A., Muñoz, C.: A formally verified generic branching algorithm for global optimization. In: Revised Selected Papers of VSTTE 2013. Volume 8164 of Lecture Notes in Computer Science., Springer (2013) 326–343
16. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Proceedings of CADE 1992. Volume 607 of Lecture Notes in Artificial Intelligence., Springer (1992) 748–752

17. Magron, V., Constantinides, G., Donaldson, A.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. **43**(4) (2017) 34:1–34:31
18. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival: The ASTREÉ Analyzer. In: Proceedings of the 14th European Symposium on Programming (ESOP 2005). Volume 3444 of Lecture Notes in Computer Science., Springer (2005) 21–30
19. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Proceedings of the 13th European Symposium on Programming Languages and Systems, ESOP 2004. Volume 2986 of Lecture Notes in Computer Science., Springer (2004) 3–17
20. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS 2008. Volume 5356 of Lecture Notes in Computer Science., Springer (2008) 3–18
21. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Proceedings of the 5th International Symposium on Static Analysis (SAS '98). (1998) 200–214
22. Martel, M.: Semantics of roundoff error propagation in finite precision calculations. Higher-Order and Symbolic Computation **19**(1) (2006) 7–30
23. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Proceedings of VMCAI 2011. Volume 6538 of Lecture Notes in Computer Science., Springer (2011) 232–247
24. de Figueiredo, L.H., Stolfi, J.: Affine arithmetic: Concepts and applications. Numerical Algorithms **37**(1-4) (2004) 147–158
25. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Proceedings of APLAS 2013. Volume 8301 of Lecture Notes in Computer Science., Springer (2013) 50–57
26. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: Proceedings of the 22nd International Conference on Computer Aided Verification, CAV 2010. Volume 6174 of Lecture Notes in Computer Science., Springer (2010) 212–226
27. Goubault, E., Putot, S.: Perturbed affine arithmetic for invariant computation in numerical program analysis. CoRR **abs/0807.2961** (2008)
28. Martel, M.: RangeLab: A static-analyzer to bound the accuracy of finite-precision computations. In: Proceedings of SYNASC 2011, IEEE Computer Society (2011) 118–122
29. Harrison, J.: HOL light: An overview. In: Proceedings of TPHOLs 2009. Volume 5674 of Lecture Notes in Computer Science., Springer (2009) 60–66
30. Ramananandro, T., Mountcastle, P., Meister, B., Lethin, R.: A unified coq framework for verifying C programs with floating-point computations. In: Proceedings of CPP 2016, ACM (2016) 15–26
31. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM Transactions on Programming Languages and Systems **39**(2) (2017) 8:1–8:28
32. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects of Computing **27**(3) (2015) 573–609

Author Index

Al Ameen, Mahmudul Faisal 430
Aldrich, Jonathan 25
Aminof, Benjamin 1
Armstrong, Alasdair 183

Bader, Johannes 25
Bansal, Sorav 161
Baumann, Cedric 47
Biondi, Fabrizio 71
Bourgerie, Austin 269
Bruni, Roberto 94
Buro, Samuele 116
Butler, Eric 138

Chin, Wei-Ngan 430

Dahiya, Manjeet 161
Dan, Andrei Marian 47
Dongol, Brijesh 183

Enescu, Michael A. 71
Eugster, Patrick 358

Feliú, Marco A. 516
Fiedor, Tomáš 205
Foster, Jeffrey S. 269

Giacobazzi, Roberto 94
Gori, Roberta 94

Heuser, Annelie 71
Hoefler, Torsten 47
Holík, Lukáš 205
Humenberger, Andreas 226

Itzhaky, Shachar 382

Jacobs, Swen 247
Jagadeesan, Radha 183
Jaroschek, Maximilian 226
Jovanović, Dejan 495

Kazerounian, Milod 269
Kovács, Laura 226
Kranz, Julian 291

Lee, Benedict 430
Legay, Axel 71
Li, Yong 313

Mastroeni, Isabella 116
McMillan, Kenneth L. 336
Meel, Kuldeep S. 71
Meshman, Yuri 47
Moscato, Mariano 516
Muñoz, César A. 516

Najafzadeh, Mahsa 358

Peleg, Hila 382, 406
Popović, Zoran 138
Prabawa, Adi 430

Quilbeuf, Jean 71

Ranzato, Francesco 452
Rasin, Dan 406
Riely, James 183
Rogalewicz, Adam 205
Roohi, Nima 474
Rubin, Sasha 1

Sakr, Mouhammad 247
Schewe, Sven 313
Schindler, Tanja 495
Shapiro, Marc 358
Shoham, Sharon 382
Simon, Axel 291
Sinn, Moritz 205
Stoilkovska, Ilina 1

Tanter, Éric 25
Titolo, Laura 516

Torf, Jordan 336
Torlak, Emina 138, 269
Turrini, Andrea 313

Vazou, Niki 269
Vechev, Martin 47
Viswanathan, Mahesh 474
Vojnar, Tomáš 205

Widder, Josef 1

Yahav, Eran 406

Zhang, Lijun 313
Zuck, Lenore D. 336
Zuleger, Florian 1, 205