Alexandra-Olimpia Bugariu

Automatically Identifying Soundness and Completeness Errors in Program Analysis Tools

Diss. ETH No. 28269

DISS. ETH NO. 28269

AUTOMATICALLY IDENTIFYING SOUNDNESS AND COMPLETENESS ERRORS IN PROGRAM ANALYSIS TOOLS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

ALEXANDRA-OLIMPIA BUGARIU

M.Sc., Technische Universität Kaiserslautern, Germany and
Laurea Magistrale in Computer Science, Free University of
Bozen-Bolzano, Italy

born on 12.01.1990

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Anders Møller, co-examiner
Prof. Dr. Cesare Tinelli, co-examiner
Prof. Dr. Zhendong Su, co-examiner

2022

To Mirela and Marcel

ABSTRACT

Program analysis tools (such as static analyzers, SMT solvers, and program verifiers) are extremely important for ensuring the correctness of a large variety of software systems. Very often, these tools are assumed to be *sound* (i.e., do not miss errors) and *complete* (i.e., have a low rate of false positives), otherwise their results are not reliable. However, these assumptions do not always hold in practice. Even if their theoretical designs have been proven correct, the actual implementations can still contain issues. We thus propose through this dissertation systematic techniques for automatically identifying soundness and completeness errors in the implementations of program analysis tools. Other types of bugs, such as performance or convergence issues, can be also detected as by-products.

Our first contribution is a novel combination of automatic test case generation approaches for identifying soundness, precision, and termination issues in the implementations of numerical abstract domains, the main components of static analyzers based on abstract interpretation. We show that our technique effectively detects errors in widely-used libraries for numerical analyses, outperforming dynamic symbolic execution and grey-box fuzzing. Our work applies also to abstract domains that rely on machine learning to improve the performance of the analysis.

Our second contribution is an automated approach for synthesizing SMT formulas that are satisfiable or unsatisfiable by construction. Together with the known ground truth, these are used to test the implementations of SMT solvers. We generate satisfiable formulas together with models, and unsatisfiable formulas together with unsat cores; being incrementally complex, they facilitate debugging and faster error localization. We evaluated our work on three widely-used SMT solvers, Z3-seq, Z3str3, and CVC4 and on the automata-based solver MT-ABC. Our experimental results show that our approach effectively detects soundness, performance, completeness, and precision problems. It is applicable also to MAX-SMT solvers.

Our third contribution is an automated technique that allows the developers to detect completeness issues in SMT-based program verifiers and soundness errors in their axiomatizations. Moreover, our approach helps them devise better triggering strategies for all future runs of their tool with E-matching. We developed a novel algorithm for synthesizing the triggering terms necessary to complete unsatisfiability proofs using E-matching. We evaluated our work on benchmarks with known triggering issues from four program verifiers. Our experiments show that it successfully synthesized the missing triggering terms in the majority of the cases, and can significantly reduce the human effort in localizing and fixing the errors.

ZUSAMMENFASSUNG

Werkzeuge zur Programmanalyse (wie etwa Static Analyzer, SMT-Solver und Programmverifizierer) sind besonders wichtig, um die Korrektheit einer Vielzahl von Softwaresystemen sicherzustellen. Sehr oft wird davon ausgegangen, dass diese Tools *korrekt* sind (d.h. keine Fehler werden übersehen) und *vollständig* sind (d.h. eine geringe Anzahl falsch positiver Ergebnisse werden produziert). Ansonsten sind die Ergebnisse nicht zuverlässig. Diese Annahmen sind jedoch in der Praxis nicht immer erfüllt. Selbst wenn sich die theoretischen Grundlagen als richtig herausstellen, können die tatsächlichen Implementationen immer noch Fehler enthalten. Wir schlagen daher in dieser Dissertation systematische Techniken zur automatischen Identifizierung von Korrektheits- und Vollständigkeitsfehlern in den Implementationen von Programmanalysewerkzeugen vor. Andere Arten von Fehlern, wie Leistungs- oder Konvergenzprobleme, können ebenfalls als Nebenprodukte erkannt werden.

Unser erster Beitrag ist eine neuartige Kombination von Ansätzen zur automatischen Testfallgenerierung zur Identifizierung von Korrektheits-, Präzisions- und Terminierungsproblemen in den Implementationen von numerischen abstrakten Domänen, den Hauptkomponenten von Static Analyzers, die auf abstrakter Interpretation basieren. Wir zeigen, dass unsere Technik Fehler in weit verbreiteten Bibliotheken für numerische Analysen effektiv erkennt und sowohl die dynamische symbolische Ausführung als auch Grey-Box-Fuzzing übertrifft. Unsere Technik ist auch auf abstrakte Domänen anwendbar, die maschinelles Lernen anwenden, um die Performance der Analyse zu verbessern.

Unser zweiter Beitrag ist ein automatisierter Ansatz zur Synthese von SMT-Formeln, die aufgrund ihrer Konstruktion erfüllbar oder unerfüllbar sind. Die SMT-Formeln zusammen mit ihren bereits bekannten Grundwahrheiten werden verwendet, um die Implementationen von SMT-Solvern zu testen. Wir generieren erfüllbare Formeln zusammen mit ihren entsprechenden Modellen und unerfüllbaren Formeln zusammen mit ihren unerfüllbaren Kernen; da sie inkrementell komplex sind, erleichtern sie das Debuggen und ermöglichen eine schnellere Fehlerlokalisierung. Wir haben unsere Arbeit an drei weit verbreiteten SMT-Solvern, Z3-seq, Z3str3 und CVC4, und am automatengestützt Solver MT-ABC evaluiert. Unsere experimentellen Ergebnisse zeigen, dass unser Ansatz Probleme in Bezug auf Korrektheit, Leistung, Vollständigkeit und Präzision effektiv erkennt. Er ist auch auf MAX-SMT-Solvern anwendbar.

Unser dritter Beitrag ist eine automatisierte Technik, die es den Entwicklern ermöglicht, Vollständigkeitsprobleme in SMT-basierten Programmverifizierern und Korrektheitsfehler in ihren Axiomatisierungen zu erkennen. Darüber hinaus ermöglicht unser Ansatz, bessere Trigger Strategien für alle zukünftigen Läufe des Tools mit E-Matching herzuleiten. Wir haben einen neuartigen Algorithmus ent-

wickelt, um Trigger Terme zu synthetisieren, die notwendig sind, um Unerfüllbarkeitsbeweise unter Verwendung von E-Matching zu vervollständigen. Wir haben unsere Arbeit an Benchmarks mit bekannten Trigger Problemen von vier Programmverifizierern bewertet. Unsere Experimente zeigen, dass der Algorithmus die fehlenden Trigger Terme in den meisten Fällen erfolgreich synthetisiert und den menschlichen Aufwand zur Lokalisierung und Behebung der Fehler erheblich reduzieren kann.

THESIS PUBLICATIONS

This dissertation is based on the following publications:

Automatically Testing Implementations of Numerical Abstract Domains [39]
Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, Peter Müller ASE'18

Automatically Testing String Solvers [37]
Alexandra Bugariu, Peter Müller ICSE'20

Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers [36, 38]
Alexandra Bugariu, Arshavir Ter-Gabrielyan, Peter Müller FM'21

ACKNOWLEDGEMENTS

The PhD has been the hardest, but also the most rewarding learning experience I have had so far. I am very grateful that throughout this journey, I had the chance to meet, work with, and learn from so many great people. Without your invaluable contributions and support, I would have not been here today. Thank you so much!

It all started with Prof. Peter Müller, who offered me the opportunity to join his research group. An opportunity that changed my life. Thank you, Peter, for helping me grow and for showing me, through your own example, what a successful academic career really means: passion for research, great teaching and mentoring skills, empathy, respect, kindness, perseverance, patience, and work-life balance.

It then ended with a defense. I am very grateful to Prof. Peter Müller, Prof. Anders Møller, Prof. Cesare Tinelli, and Prof. Zhendong Su for being part of my committee and for taking the time to review my thesis. All the discussions and suggestions helped me improve the final version of this dissertation, but most importantly, they made me see various problems from different angles. These new perspectives are very valuable for choosing additional research directions to explore in the future.

In between, there were 5.5 years and many, many people who shaped my path.

My friend and at the beginning office mate, now Prof. Gagandeep Singh, is one of the people from whom I learned the most during my PhD. Thank you, Gagandeep, for teaching me abstract interpretation, for your amazing patience, and for your permanent support. From our discussions (at and outside ETH), I learned not to be ashamed to admit that I do not know, to ask questions, and to be grateful that I am surrounded by so many extremely smart people who are willing to help me.

Prof. Alexander Summers and his course on Program Verification set the foundations for two chapters of my thesis. The lecture slides were also extremely helpful for many of the student projects that I have advised. Thank you, Alex, for all your explanations, for your kindness, patience, and encouragements.

I am also very grateful to Dr. Pavol Bielik, who taught me one of the most valuable lessons: that the PhD student is the only person responsible for the success of their PhD. Thank you, Pavol, for helping me become independent and for showing me that hard work, kindness, and perseverance are so necessary for this journey. Thank you for challenging my ideas, for teaching me to put them into perspective, and for encouraging me to attend talks, volunteer for conferences and committees, and meet other researchers. I would have not joined and presented my work at the

Heidelberg Laureate Forum without your suggestion. Also thank you for showing me that presentations and Q&As can be both technically strong and very relaxed!

Learning to write papers was not easy, but Peter was a great mentor. Many thanks also to Prof. Maria Christakis and Dr. Valentin Wüstholz for all their patience and support with my first publication. I am also very grateful to Dr. Arshavir Ter-Gabrielyan, who helped me transform his idea of a joint project into a paper. Thank you, Arshavir, also for your friendship and for so many great memories!

One of the most rewarding parts of my PhD was working with students. I am very grateful to Dr. Caterina Urban for showing me how to be a good adviser. Thank you, Caterina, for all your support, for your personal example of high-quality work, great presentations, and very interesting research topics. I am also deeply thankful to Dr. Malte Schwerhoff, who, besides others, helped me during a very difficult period of my life by taking over a project on very short notice. Moreover, I was fortunate to collaborate with students with very different backgrounds, who taught me that being a mentor is also about patience, kindness, tolerance, and empathy. Thank you, Madelin Schumacher, Radwa Sherif Abdelbar, Pascal Strebel, Olivier Becker, Madalina Hurmuz, Kevin Thommen, and Sebastian Kühne for all your effort, all your questions, and all the lessons you helped me learn.

I would have not been here today without the permanent support of an amazing research group, based not only on strong technical knowledge but also on mutual respect and empathy. Thank you, Marco Eilers and Jérôme Dohrau for being great friends and office mates and for helping me find solutions for my (not always university-related) problems. I am also very grateful to Vytautas Astrauskas, who during the hardest period of my PhD, made sure I am all right. Thank you, Vytautas, for being there for me whenever I needed and for bringing Scala to our group! Many thanks also to Federico Poli, Aurel Bílý, Thibault Dardinier, and João Pereira, who, together with Marco, have shared with me one of the most challenging, but extremely pleasant experiences of my PhD: teaching the exercise sessions for Concepts of Object-Oriented Programming. I really appreciate your help, the countless discussions, and all the time and effort you invested in the exercises and the exams. I am also very grateful to Felix Wolf and Gaurav Parthasarathy for all the discussions and all their input about research, teaching, and supervision. I am sure these will be very helpful also for my future academic career. Many thanks to Linard Arquint (who, together with Felix, translated the abstract into German) and Jonáš Fiala, for their help with dry runs and for organizing presents and events, and to my new office mate, Dionisios Spiliopoulos, for being part of our group. I was very fortunate to also work with and learn from former members of the Programming Methodology group: Dr. Dimitar Asenov, Dr. Lucas Brutschy, Dr. Martin Clochard, Prof. Christoph Matheja, and Dr. Wytse Oortwijn. Thank you all!

I am also thankful to Prof. Thomas Gross, Prof. Markus Püschel, and Prof. Martin Vechev, to my colleagues from the other research groups, and to the student TAs with whom I have collaborated or discussed during my PhD. Thank you for sharing your research or teaching ideas, for your advice, feedback, and help.

I am also deeply thankful to Mrs. Marlies Weissert and Mrs. Sandra Schneider for their invaluable support throughout these years. Thank you for your kindness and willingness to help, for being so positive, and for organizing so many great events!

My dream to become a professor has been always supported by my Bachelor adviser, Prof. Marius Minea, who encouraged me to apply for internships and work on research-oriented projects. Prof. Minea, Dr. Daniel Ratiu (who introduced me to SMT solvers and the Marktoberdorf Summer school), and Prof. Sandu Popescu had a significant contribution to my decision to apply for a PhD at ETH. Thank you so much for all your advice, for the countless discussions, and invaluable feedback. Many thanks also to my former colleagues from itemis, Kolja Dummann, Dr. Domenik Pavletic, and Dr. Tamás Szabó, for all their help during the interviews.

I am also deeply grateful to my parents for their unconditional support. You taught me the importance of education and self-confidence, you taught me to dream, and to work hard to transform my dreams into reality. Studying abroad and doing my PhD at ETH were initially just dreams. Thank you for believing in me!

Since every end is a new beginning, I would like to also thank you, little one, for being with me while submitting the thesis and during the defense. Most likely, our new journey will be harder than the PhD, but I am sure we will learn together :).

CONTENTS

1

INTRODUCTION

Static analyzers and SMT-based program verifiers are valuable tools for increasing the reliability of a large variety of software systems. They can detect errors in early stages of development, by statically analyzing the behavior of a program without running it or by verifying if the program fulfills its formal specifications. For example, the analyzers Astrée [30], Clousot [73], and SLAM [13] found actual bugs in industrial code; the Dafny verifier [106] is used by Amazon Web Services [2] for proving the correctness of security-critical libraries and the Corral verifier [101] is the main verification engine of the Static Driver Verifier from Microsoft [100].

Numerous static analyzers and SMT-based program verifiers (collectively called *program analysis tools* in this dissertation) have been already built or synthesized, for different programming languages [3, 7, 13, 15, 30, 45, 66, 73, 91, 101, 106, 163]. These tools are very powerful, but also very complex pieces of software. To provide reliable results, it is thus crucial for them to be *correct*. Even if certain theoretical properties of their *design* have been formally proved, significantly fewer techniques are available for checking the correctness of their actual *implementation*.

There are very few formally verified analyzers or verifiers. Verasco [93] (a static analyzer based on abstract interpretation [51]) is an exception, as its soundness has been proved in Coq [133]. However, it required a significant proof effort (≈34 000 lines), which "involve[d] large case analyses and difficult proofs over integer and F.P. arithmetic" [93]. Verasco's example shows that it is very hard to apply formal methods to program analysis tools [4, 40]. Other techniques are thus necessary.

Our goal. This dissertation aims to fill this gap, by *developing systematic approaches for identifying errors in the implementation of program analysis tools*. We also include here SMT solvers, as they are used not only by SMT-based program verifiers, but also by techniques for program synthesis, symbolic execution, and concolic testing.

When checking the correctness, we consider two main properties: *soundness*, i.e., the tools should not miss errors (or for SMT solvers, if there exists a model for an input formula, they should classify it as satisfiable, otherwise they should report it is unsatisfiable), and *completeness*, i.e., they should have a small false positives rate. In the context of static analyzers, completeness is often also called *precision*. *Performance* and *termination* (*convergence*) issues can be also detected as by-products.

1.1 STATE-OF-THE-ART

Various approaches have been proposed in the literature for identifying errors in software systems or for automatic test case generation; some target particular types of program analysis tools. In the following, we discuss them in more detail.

Random testing. The simplest way of testing static analyzers is with randomly generated programs [56]. This approach was used to find errors in Frama-C [95] with programs randomly generated by Csmith [184]. However, since it is a black-box testing technique, it does not take into account the actual implementation of the static analyzer. Thus, it cannot guarantee a high coverage and its effectiveness depends on the ability to randomly generate programs that exercise the execution paths that are more error-prone. Moreover, this approach can produce inputs that cause the analyzer to crash, but it cannot be applied for identifying specific classes of errors, such as soundness or precision bugs, as it does not have a test oracle.

Systematically checking lattice properties. Midtgaard and Møller's work [120] improves over random testing by providing a systematic way of quickchecking [49] algebraic properties (e.g., reflexivity, associativity, commutativity, absorption, etc.). This approach was designed for static analyzers that use functions operating over lattices. Its main drawback is that it relies on generators for producing the inputs. For properties based on exhaustive checks (such as the bottom element of the lattice is the greatest lower bound of all the elements) this idea cannot be practically applied. As the authors acknowledge [120], even for simple abstract domains like Intervals [53], this test would require generating all the pairs of, e.g., 32-bit integers, which is not feasible. The same limitations apply to other precision tests.

Differential testing. Other classes of techniques were inspired from compiler testing. Differential fuzzing (cross-checking) [40] is an approach that addresses the oracle problem by comparing the output of two different analyzers, SMT solvers, or program verifiers, for the same input. Any difference in the results signals an error, but requires manual inspection to detect which of the two implementations is at fault. Moreover, this approach assumes that there are at least two analysis tools available and that their outputs are comparable, which is often not the case [40].

Equivalence Modulo Inputs (EMI) [103] provides a solution to this problem. Instead of validating two program analysis tools against each other, EMI's core idea is to perform semantically preserving transformations on the input program. These transformations can be: removing code that is never executed, inserting additional instructions with no side effects, or modifying the program such that the newly created one is equivalent, with respect to a given input, to the original. If the tool returns different results for the two programs, then it contains a bug. This technique was originally designed for validating compilers [103], but it was also successfully applied for checking the correctness of learned static analyzers [26].

A slightly different approach, also designed for compilers, is skeletal program enumeration [193]; it exhaustively generates all the programs (up to a fixed size) that can be obtained from a given syntactic skeleton by enumerating all the possible variable usage patterns. This technique can be automatically applied only for finding compiler crashes, because without manual effort, it cannot be determined which is the expected compiled code for each enumerated program. The approach was recently adapted for testing SMT solvers [186], using metamorphic relations [164] as the test oracle. However, this work (like many others in this

area [6, 34, 35, 126] or for testing static analyzers [4]) require delta debugging [189] to minimize the inputs that exposed the issues and to facilitate error localization.

(Dynamic) symbolic execution (DSE). A large palette of approaches also covers the area of automatic test case generation. As opposed to random testing, symbolic execution engines use the structure of the tool under test to generate the test cases. The programs are evaluated with symbolic variables that abstract classes of inputs. When a conditional expression is encountered, the execution conceptually forks, since without concrete values, both branches can be equiprobably explored. A boolean formula is then constructed for each execution path, cumulating the corresponding constraints over the symbolic variables. An SMT solver is used to check the satisfiability of the formula and to generate, if they exists, concrete instantiations of the symbolic variables, which represent the test cases.

One of the disadvantages of symbolic execution is that it tries to explore all the possible paths, which is not always feasible in practice. If the programs contain loops that depend on symbolic variables, the number of branches is potentially infinite [12]. Moreover, SMT solvers are incomplete, thus they cannot provide answers for all the formulas. For example, some SMT solvers cannot handle floating point arithmetic. Different solutions have been proposed for those cases, such as unconstrained mathematical optimization (implemented in XSat [76]) or Darulova and Kuncak's technique for the sound compilation of reals [60].

Other ways of handling the limitations of random testing and symbolic execution are provided by white-box fuzzing [81] (also called coverage-guided fuzzing) and DSE. These techniques combine symbolic and concrete executions; therefore, they have access to concrete values even when the SMT solver cannot produce them. Furthermore, they drive the execution towards unexplored paths, by negating terms of the previous path constraint, increasing the coverage, and avoiding the generation of redundant inputs. Numerous DSE tools are currently available, for different programming languages: KLEE [41], DART [80], CUTE [143] for C (LLVM [102]), PEX [173] for C#, jCUTE [142] and Java PathFinder [178] for Java, are just a few examples. A complete list can be found in [12]. To the best of our knowledge, though, (D)SE has not been directly applied to analyzers or verifiers.

Despite its aforementioned advantages, DSE also has its own limitations. A major problem in practice is that it cannot handle calls to external libraries when the parameters are symbolic variables. In those cases DSE applies concretization, i.e., it replaces the symbolic variables with concrete values, obtained by solving the current path constraint. Due to this simplification, DSE loses completeness, leaving some execution paths unexplored [43]. Synthesizing framework models that can be symbolically executed [92] could be an alternative to concretizing, but this approach is currently available only for subsets of Swing and Android.

Specification-based testing. Other techniques use formal specifications for automatic test case generation. Korat [33], for example, is a framework for testing Java applications, which derives the input data from the method's precondition (expressed in JML [105]) and uses the postcondition as the test oracle. Nevertheless,

this approach can be applied only when method contracts and class invariants are provided for the entire source code, which rarely happens in practice for complex software, such as static analyzers, SMT solvers, or program verifiers.

1.2 CHALLENGES

Designing techniques for identifying errors in program analysis tools is not trivial. Testing software developed for ensuring the correctness of other pieces of software is different than searching for issues in regular programs. All the approaches presented in Section 1.1 have their own advantages, but they have also limitations. In this dissertation, we aim to create new techniques, to redesign, or to combine the existing ones for reliably finding soundness and completeness errors in the implementations of different types of program analysis tools. To achieve our goal, we have to overcome multiple, interconnected challenges, derived from open research problems. We discuss them next, together with possible solutions.

Input data. An essential part of any automated testing approach consists in generating the input data. While static analyzers and program verifiers accept programs as inputs, these may not provide the optimal level of granularity for soundness and completeness testing and for fast error localization (as we have seen for random testing [56] in Section 1.1). The alternative to considering the static analyzer or the program verifier as a whole is to generate inputs only for those critical components that are more error-prone (e.g., for the abstract domains, as in [120]). Moreover, to overcome the limitations of approaches like [120], our scope is to construct representative, increasingly complex, and diverse inputs automatically, without relying on user-provided generators.

Test oracles. As we have explained in Section 1.1, many of the existing techniques do not have a test oracle, thus they cannot be used for automatically finding soundness bugs, without additional manual effort. To address the limitations of, e.g., differential testing [40], another expected property of our work is to identify bugs without having a reference implementation of the tool under test. This constraint is often found in practice, as there might not exist alternative, trusted implementations. It is thus important that we construct test oracles that provide guarantees about the type of the revealed errors (unsoundness, incompleteness, etc.).

Easy error reporting. The goal of testing is to increase the reliability of a system. The bugs found by our approach should thus be easy to understand, reproduce, and report. Various state-of-the-art techniques (see Section 1.1) rely on external tools for minimizing the failed test cases. However, while these produce syntactically small inputs, they might not always represent the simplest, most intuitive counterexamples developers would write manually. We thus aim to construct increasingly complex inputs, such that the issues are exposed by simple ones, avoiding minimization and duplicated bug reports (i.e., with the same root cause).

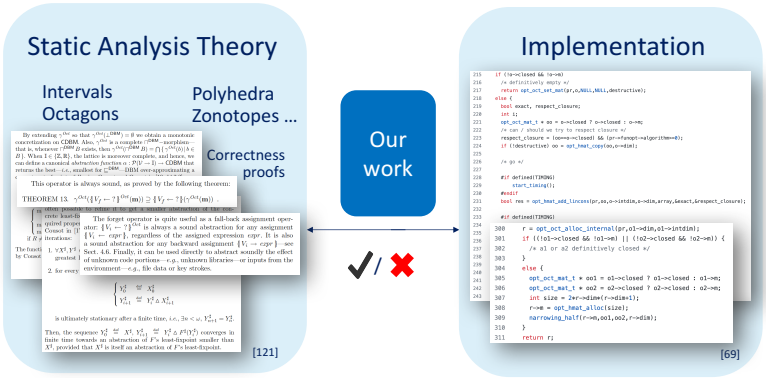

FIGURE 1.1: High level overview of our work from Chapter 2, which checks if the implementation of numerical abstract domains fulfills the theoretical guarantees.

Relevance. A related aspect is relevance, which influences the response of the developers to the reported issues. The errors identified by our technique should be relevant for the *developers* of the program analysis tools (i.e., should represent bugs they will consider fixing), but also for their *users* (i.e., should occur in realistic usage scenarios, and not be caused, e.g., by experimental combinations of options).

Fast error localization. Our work should help the developers identify the root cause of the errors. This can be achieved by generating minimal test cases.

Usability. To facilitate their adoption in practice, our tools should be easy to use and integrate into the development process.

Applicability. Our approach is also expected to generalize to other use cases, besides the ones it was designed for. A technique for testing SMT solvers, for example, should be also applicable to other provers that accept SMT-LIB [18] inputs.

1.3 THIS DISSERTATION

In the following, we briefly introduce the three techniques we developed for identifying soundness and completeness errors in program analysis tools; these address the challenges from Section 1.2. The details are explained in Chapter 2, Chapter 3, and Chapter 4. Chapter 5 presents possible future research directions.

Our techniques. Figure 1.1 gives a high level overview of our approach for testing static analyzers based on abstract interpretation (Chapter 2). Our work checks if the actual implementations of numerical domains (the main components of static numerical analyses) fulfill their theoretical guarantees, i.e., are sound, precise, and eventually converge (reach a fixed point). For this, we derived 46 general properties that they should satisfy and used them as the test oracles. We also developed

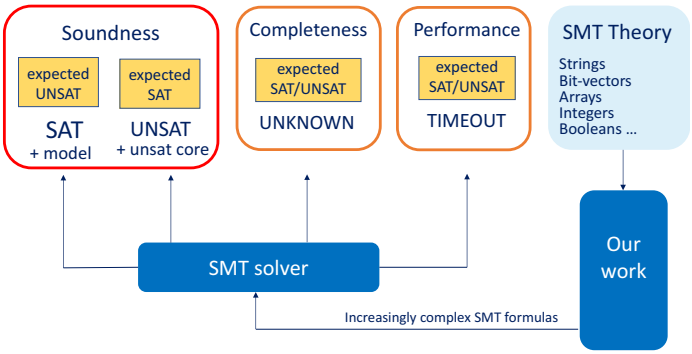

FIGURE 1.2: High level overview of our work from Chapter 3, which synthesizes increasingly complex formulas with known ground truth for testing SMT solvers.

a novel algorithm for constructing the input data, by applying a sequence of domain operations. Our experiments show that our work can find bugs in widely-used numerical domains, outperforming fuzzing and DSE. Our technique is also applicable to more complex domains, which use machine learning algorithms to improve the performance of the analysis.

Figure 1.2 depicts our approach for synthesizing SMT formulas that are satisfiable or unsatisfiable by construction; these can be used for testing SMT, as well as automata-based solvers (Chapter 3). We also generate models or minimal unsat cores and consider them as additional oracles. We start with simple formulas and then derive more complex ones, by applying a set of satisfiability-preserving transformations that we defined. In this way, most of the bugs are exposed by simple inputs, which facilitates debugging and error localization. All our generated benchmarks are publicly available [129]. As the experiments show, our technique can find soundness and completeness errors in state-of-the-art solvers, which cannot be identified by the closest-related fuzzing work. Our approach can be also applied for testing MAX-SMT solvers, by encoding the transformations as hard or soft constraints, with predefined weights.

Figure 1.3 presents our approach for enabling the developers of SMT-based program verifiers identify and fix incompleteness errors in their implementation and unsoundness in their axiomatizations (Chapter 4). Both problems occur when E-matching [63] (the prevalent SMT algorithm for verification benchmarks) cannot complete an unsatisfiability proof due to missing quantifier instantiations (caused by overly restrictive patterns). We, therefore, propose a novel algorithm that augments E-matching and synthesizes the triggering terms required to perform the missing instantiations. Our experimental results show that our work is effective on inputs produced by a diverse set of mature program verifiers. We also present an alternative approach for extracting the same information from refutation proofs.

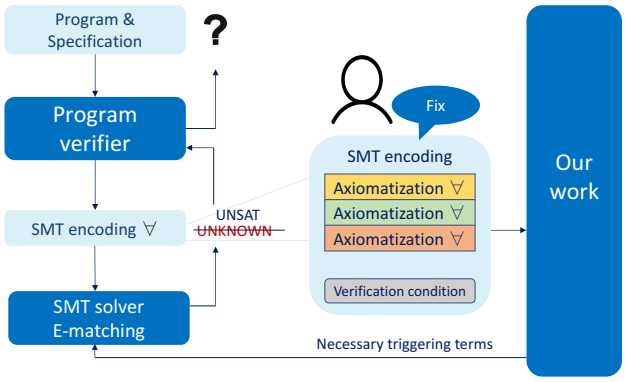

FIGURE 1.3: High level overview of our work from Chapter 4, which synthesizes the triggering terms necessary for E-matching to complete unsatisfiability proofs.

Contributions. The main contributions of this dissertation are summarized below:

- Chapter 2: presents our novel combination of automatic test case generation techniques for identifying soundness, precision, and termination issues in the implementations of abstract domains. We demonstrate that our approach effectively detects both seeded and real errors in widely-used libraries for numerical analyses, outperforming DSE and grey-box fuzzing.

- Chapter 3: describes our automated approach for synthesizing SMT formulas for the string theory, which are satisfiable or unsatisfiable by construction. Together with the known ground truth, these formulas are used to automatically test the implementations of SMT solvers. Our technique generates satisfiable formulas together with models, and unsatisfiable formulas together with unsat cores; being incrementally complex, they facilitate debugging and faster error localization. We implemented our technique and evaluated it on three widely-used SMT solvers, Z3-seq [28, 187], Z3str3 [21], and CVC4 [112] and on the automata-based solver MT-ABC [9]. Our experimental results show that our approach effectively detects soundness problems and outperforms the closest fuzzing technique for string solvers in doing so. Our work can also reveal other types of errors, such as performance, completeness, or precision issues. Moreover, it generalizes to other theories and their combinations and is applicable also to MAX-SMT solvers.

- Chapter 4: introduces the first automated technique that allows the developers detect completeness issues in program verifiers and soundness problems in their axiomatizations. Moreover, our approach helps them devise better triggering strategies for *all* future runs of their tool with E-matching. We developed a novel algorithm for synthesizing the triggering terms necessary

> to complete unsatisfiability proofs using E-matching. Since quantifier instantiation is undecidable for first-order formulas over uninterpreted functions, our algorithm might not terminate. However, all identified triggering terms are indeed sufficient to refute the formulas, i.e., there are no false positives. We evaluated our technique on benchmarks with known triggering problems from four program verifiers. Our experimental results show that it successfully synthesized the missing triggering terms in 65.6% of the cases, and can significantly reduce the human effort in localizing and fixing the errors.

Chapter 2 is mainly based on our ASE'18 paper [39]. The technique from Chapter 3 was introduced in our ICSE'20 paper [37], while the extensions were partly explored in the context of student projects [20, 99, 172]. Chapter 4 presents the algorithm from our FM'21 paper [38], as well as additional examples and optimizations from its extended version [36]; the alternative approach for constructing triggering terms from unsatisfiability proofs builds on a student project [157].

2

IDENTIFYING SOUNDNESS AND PRECISION ERRORS IN NUMERICAL ABSTRACT DOMAINS

In this chapter, we present our technique for automatically testing the implementations of numerical abstract domains, which are the main components of state-of-the-art static analyses. We also show how our approach can be applied for testing domains that use machine learning for achieving better performance.

2.1 INTRODUCTION

Static program analyses compute semantic properties of programs, which are the basis for optimizations and for identifying errors and security vulnerabilities. Since most program properties are undecidable, static analyses approximate the set of possible behaviors. For their results to be reliable and useful in practice, the analyses should be *sound* and *precise*. A *sound* analysis considers each possible program behavior; to capture them, it computes an over-approximation of all possible behaviors. A sound analysis should therefore not produce false negatives. A *precise* analysis should compute a tight over-approximation, to minimize false positives.

Many static analyses are based on the abstract interpretation framework [51], in which the program state is represented by elements of *abstract domains*. For instance, numerical abstract domains can track intervals of possible values for numerical variables or constraints between them. The semantics of program operations is represented by *abstract transformers*, which specify the effect of an operation on the abstract state. Even though it is possible to prove properties of the *design* of a static analysis, ensuring soundness and precision for its *implementation* is challenging. This is even harder for implementations of abstract domains, which are often complex and highly optimized, to maximize performance and scalability [148]. Errors in these implementations likely affect the usefulness of all static analyzers that build on them, thus it is extremely important to detect them reliably.

Let us consider a static analysis that abstracts numerical variables as intervals of possible values. The abstract value $[0, 5]$ for an integer variable x expresses that, in each program execution, the concrete (actual) value of x satisfies the constraint $0 \leq x \leq 5$. Figure 2.1 illustrates a potential unsoundness due to arithmetic overflow. Without prior knowledge about the parameter `p` and assuming that the abstract domain is implemented using bounded integers, the abstract value of `p` is $[INT\_MIN, INT\_MAX]$; thus the abstract value of `a` after the assignment is $[INT\_MIN + 1, INT\_MAX + 1]$. A naive implementation of the addition operator may lead to overflow and produce $[INT\_MIN + 1, INT\_MIN]$. This empty interval indicates that the code after the assignment is unreachable, which is unsound. The unsoundness might mask errors and security vulnerabilities in subsequent code.

```
void foo(int p) {
  int a := p + 1; // p -> [INT_MIN, INT_MAX]
  ...     // a -> [INT_MIN + 1, INT_MAX + 1] = [INT_MIN + 1, INT_MIN]
}
```

FIGURE 2.1: Potential unsoundness due to overflow. The abstract state, captured by the Intervals domain implemented with bounded integers, is shown as comments.

This work. In this chapter, we present an automatic testing technique for identifying *soundness* errors (like the one in Figure 2.1), as well as *precision* and *termination* issues in widely-used implementations of abstract domains. We generate test inputs using a novel combination of existing ideas: starting from a set of predefined values, we apply abstract-domain operations to create representative domain elements, and vary the operations by employing an off-the-shelf grey-box fuzzer, such as AFL [166] or LibFuzzer [113], to maximize the coverage of the tested implementation. As in earlier work by Midtgaard and Møller [120], we use mathematical properties of abstract domains as test oracles. However, we target real-world implementations of complex abstract domains (e.g., APRON's Octagons domain [121] and ELINA's Polyhedra domain [148]), and extend the set of tested properties by including more precision properties and by approximating termination properties.

Our evaluation on several abstract domains of the APRON [90] and ELINA [148] libraries shows that our combination of techniques effectively detects soundness and precision problems in complex, mature implementations. In particular, we show that it is more effective than purely relying on existing test case generation techniques, such as grey-box fuzzing and dynamic symbolic execution (DSE), also known as concolic testing [42, 80].

Contributions. This chapter makes the following contributions:

- We present a novel combination of automatic test case generation techniques to detect soundness, precision, and termination issues in implementations of abstract domains.
- We demonstrate that our approach effectively finds both seeded and real errors in widely-used implementations of numerical abstract domains.
- We show that our technique tests abstract domains more effectively than standard DSE or grey-box fuzzing approaches.

Our work can help the developers of abstract domains ensure their implementations are sound and precise. It is also useful for the developers of static analyzers to assess the quality of existing libraries. Even if the design of an abstract domain intentionally sacrifices soundness in favor of other qualities [46, 114], it is important to detect *unintentional* unsoundness due to implementation errors.

Outline. The rest of this chapter is structured as follows: Section 2.2 summarizes background information on abstract domains. Section 2.3 gives an overview of our

approach and Section 2.4 explains the technical details. In Section 2.5, we present our experimental evaluation on widely-used implementations of abstract domains. We illustrate how our technique generalizes to domains that combine abstract interpretation with machine learning in Section 2.6 and present its limitations in Section 2.7. We discuss related work in Section 2.8 and conclude in Section 2.9.

2.2 BACKGROUND: ABSTRACT DOMAINS

Abstract interpretation [51] is a theoretical framework for expressing static analyses, used by various industrial analyzers (e.g., Astrée [30], Clousot [73]). It relies on *abstract domains* to represent abstractions of concrete program states, and on *abstract transformers* to model the behavior of program instructions, such as assignments and conditionals. Abstract domains are often reused across different program analyses. Most static analyzers employ numerical domains, which are the focus of this work. Widely-used numerical domains include Intervals [53], which capture the range of values for each variable, Octagons [121], which can also express simple relations between two variables, Polyhedra [55], which can capture linear inequalities between arbitrarily many variables, and Zonotopes [82], which express affine relations.

Most abstract domains are represented by complete lattices of the form $(\bar{A}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$. $\bar{A}$ denotes the set of abstract elements $\bar{x}$, which are partially ordered by *inclusion* $\sqsubseteq$. Each abstract element represents a set of *constraints*, i.e., mathematical relations between variables and constants. The *bottom* element $\bot$ is the least element of the lattice; it represents the empty set of concrete states and corresponds to an unsatisfiable set of constraints. The *top* element $\top$ is the greatest element of the lattice and represents the universal set of concrete states; that is, all the variables are unconstrained. $\sqcup$ and $\sqcap$ are the *join* and *meet* operators, which are used to compute the union, respectively the intersection, of two abstract elements. Additionally, an abstract domain whose lattice has an infinite height requires a *widening* operator (∇) to ensure that the analysis eventually reaches a fixed point. Some domains (such as Intervals and Octagons) also support a *narrowing* operator ($\triangle$), which can improve the precision of the analysis [52].

Abstract transformers are typically specific to a given analysis and programming language, but some of them are universal building blocks for many analyses. These include *assign* to represent an assignment, *cond* to assume that a condition holds, and *project* to remove any previous information about a variable (e.g., when a new value is read from a file). Our work focuses on these three transformers as they are the most complex to implement [147] (and thus the most error-prone).

2.3 OVERVIEW

Since implementations of abstract domains are often used as libraries by different program analyses, we apply a *unit testing* approach. Compared to system testing,

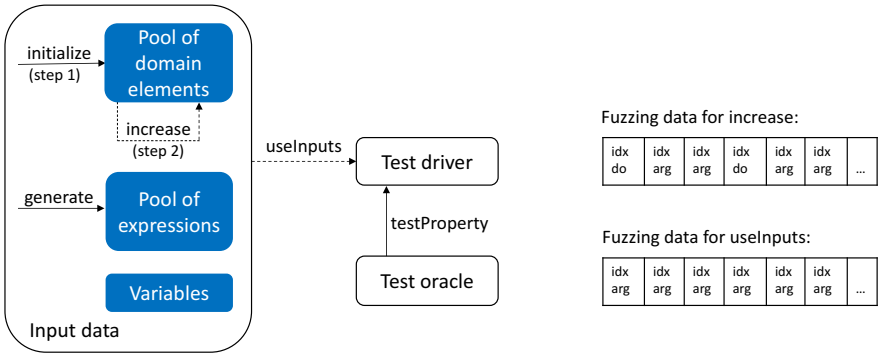

FIGURE 2.2: Left: The main components of our technique, represented as white boxes. The blue boxes illustrate different types of input data, the arrows depict actions. Those actions marked with dashed lines are performed by the fuzzer. Right: Structure of the fuzzing data, represented as an array of integers consisting of: indexes for domain operations (idx do), indexes in the pool of domain elements/expressions, or in the list of variables for their arguments (idx arg).

unit testing allows us to specify *generic* test oracles, which are independent of a specific abstract domain or static analysis. Moreover, it facilitates the construction of test data because abstract-domain elements can be generated much easier than input programs for the whole analyzer [120]. In this section, we give an overview of the three main ingredients of our automatic unit testing approach: test oracles (Section 2.3.1), input data generation (Section 2.3.2), and test drivers (Section 2.3.3). They are depicted in Figure 2.2 (left). The details follow in Section 2.4.

As opposed to a related work that requires the users to provide complex generators for the abstract-domain elements [120], we use a grey-box fuzzer to construct the input data and to choose the parameters for the property under test (see the dashed lines in Figure 2.2). Such fuzzers use a lightweight code instrumentation, which enables us to generate inputs that are likely to execute previously-uncovered code. All the fuzzing data (i.e., the values created by the fuzzer) can be encoded into one array of integers, which has the structure shown in Figure 2.2 (right).

2.3.1 *Test oracles*

The abstract interpretation framework prescribes a number of properties of the domain operators and abstract transformers (collectively referred to as *domain operations* in the rest of this chapter), which are required for soundness. For instance, if the abstract element $\overline{x}$ (capturing the pre-state of an assignment) is reachable (that is, different from bottom), the post-state of the assignment $v := e$ should also be non-bottom for all program variables v and well-formed expressions e:

$$\overline{x} \neq \bot \Rightarrow \mathit{assign}(\overline{x}, v, e) \neq \bot \tag{2.1}$$

We use these general soundness conditions as test oracles. We also identify several general precision conditions, whose violation indicates that the result of a domain operation is over-approximated more than necessary. For this purpose, we compare the result of a domain operation to the *best transformer*, that is, the most precise result *representable* in a given abstract domain. For instance, the result of intersecting top with an abstract element should be equal to the element itself:

$$\top \sqcap \overline{x} = \overline{x} \tag{2.2}$$

Moreover, we check that widening and narrowing converge within a given number of iterations, which ensures the termination of any fixed-point computation in which they are used.

Note that all the properties considered in this chapter are defined under the assumption that the analyzed programs do not raise exceptions, otherwise the behavior of the static analyzer depends on the semantics of the programming language. In C, for example, division by zero causes undefined behavior. When classifying the properties into soundness and precision (as described in Section 2.4), we assume that the abstract domain does not model error states.

2.3.2 *Input data*

Abstract domains often use sophisticated data structures to optimize performance. For example, the elements of the Polyhedra domain are typically represented using both matrices and vectors of floating-point numbers. In our experiments, we observed that the standard test case generation techniques do not work well for complex abstract domains. In particular, fuzzing failed to detect subtle interactions between domain operations, and DSE did not effectively explore real-world implementations that make heavy use of floating-point arithmetic and libraries. We thus use a combination of test case generation techniques to create a pool of domain elements, which serve as test inputs to domain operations. The pool is constructed in two steps, presented in pseudo-code in Algorithm 2.1. The parameter `fuzzingData` represents the array of indexes provided by the fuzzer (see Figure 2.2, right). The other parameters are explained in the following paragraphs.

Step 1: Initialize the pool. We first create an initial pool of a configurable size (parameter `size` in Algorithm 2.1), by combining boundary and random testing (Algorithm 2.1, line 2). Each element of a numerical domain can be constructed from numerical constraints; e.g., an element of the Intervals domain, which maps program variables v_i to their possible values, is created from the constraint $k_l \leq v_i \leq k_u$ (the constants k_l and k_u are the lower and upper bounds). We generate the constants randomly or choose them from a predefined set of boundary values that are more likely to expose bugs, such as off-by-one errors and arithmetic overflows. The parameter `predefined` in Algorithm 2.1 controls which of these two alternatives is used. For example, if the Intervals domain is implemented

Algorithm 2.1: Our algorithm for constructing the pool of abstract elements and the well-formed expressions.

Arguments : size — initial pool size
predefined — whether to use predefined values or not
nbops — number of operations used to increase the initial pool
ops — domain operations
vars — program variables
fuzzingData — indexes provided by the fuzzer

Result: The pool of domain elements and the pool of expressions

```
1  Procedure constructPool
2      pool ⟵ initialize(size, predefined)                    // step 1
3      exprs ⟵ generate(vars, predefined)
4      foreach i ∈ {0, nbops − 1} do                          // step 2
5          op ⟵ ops[nextIndex(fuzzingData)]
6          x̄ ⟵ pool[nextIndex(fuzzingData)]
7          if op ∈ {⊔, ⊓, ∇} then
8              ȳ ⟵ pool[nextIndex(fuzzingData)]
9              res̄ ⟵ x̄ op ȳ
10         else if op = assign then
11             v ⟵ vars[nextIndex(fuzzingData)]
12             e ⟵ exprs[nextIndex(fuzzingData)]
13             res̄ ⟵ assign(x̄, v, e)
14         else if op = project then
15             v ⟵ vars[nextIndex(fuzzingData)]
16             res̄ ⟵ project(x̄, v)
17         pool ⟵ pool ∪ {res̄}
18     return (pool, exprs)
```

using machine integers, the boundary values are $\{INT\_MIN, 0, INT\_MAX\}$. The initial pool also contains the extreme elements $\top$ and $\bot$.

Besides domain elements, some transformers (e.g., *assign*) also require well-formed expressions. We thus also generate a pool of expressions, which are linear sums over program variables (parameter `vars` in Algorithm 2.1); we choose their coefficients randomly or from a set of boundary values (Algorithm 2.1, line 3).

Selecting inputs from the initial pool (together with a suitable expression e) as arguments to the *assign* transformer will likely detect the possible unsoundness illustrated in Figure 2.1. The pool is likely to contain an element mapping the variable `p` to $[INT\_MIN, INT\_MAX]$, as INT_MIN and INT_MAX are predefined boundary values; moreover, we are likely to obtain an expression e of the from `p`$+k$ for some positive constant k (see Section 2.4 for details). Evaluating the *assign* transformer on these inputs violates the soundness property (2.1) from Section 2.3.1.

The initial pool allows us to test *individual* domain operations. However, this approach is insufficient in two situations. First, some implementations rely on complex consistency conditions on their data structures. For Polyhedra, for instance, the two internal representations must be kept consistent. If a faulty operation violates this invariant, the effect can be often observed only when applying a subsequent operation; it is thus necessary to perform at least two consecutive operations to detect the bug. Second, there are certain soundness or precision properties of *individual* operations that cannot be checked by generic, domain-independent oracles. For example, the *assign* transformer for Octagons should, in some cases, apply a so-called *closure* operation; failing to do so or using an imprecise closure may lead to loss of precision in subsequent operations, such as inclusion or equality tests [121]. A test oracle that directly detects a missing closure would be specific to Octagons and, thus, not reusable. A more generic way to detect this problem is by intersecting the result of the *assign* transformer with top. The expected output of the intersection is the same as the result of the *assign* transformer itself. However, if the transformer does not apply the closure when expected, the domain-independent property (2.2) from Section 2.3.1 may fail due to imprecision.

Step 2: Increase the pool. The above situations can both be addressed by executing at least two consecutive domain operations before checking the oracle. Therefore, step 2 of our input generation technique (Algorithm 2.1, lines 4–17) expands the pool of domain elements by iteratively applying a domain operation (from the list of supported operations `ops`) to existing elements (and possibly program variables and expressions — as in Algorithm 2.1, lines 11–12 and 15) and adding the result to the pool (Algorithm 2.1, line 17). The fuzzer controls which elements, expressions, and variables should be used in this step, through the parameter `fuzzingData` (see Figure 2.2, right). The procedure `nextIndex` returns the following index provided by the fuzzer. By repeating this process a configurable number of times (parameter `nbops` in Algorithm 2.1), we increase the likelihood of constructing elements that need to be built incrementally with several domain operations. We are thus more likely to detect bugs that manifest themselves only in consecutive operations.

2.3.3 *Test drivers*

For each property under test, we manually write a test driver. Algorithm 2.2 shows our test driver for checking the soundness property (2.1) for the *assign* transformer. It takes as arguments the pool of abstract elements (`pool`) and the pool of expressions (`exprs`), which were constructed according to Algorithm 2.1, a list of program variables (`vars`), and three indexes encoded into the `fuzzingData`. The driver first obtains the domain element, the variable, and the expression corresponding to the indexes (Algorithm 2.2, lines 2–4), then it applies the transformer, i.e., it executes its implementation from the tested numerical library (Algorithm 2.2, line 5). Finally, it asserts the property (2.1) on the result (Algorithm 2.2, line 6).

Algorithm 2.2: Our test driver for checking the soundness property (2.1) of the assignment transformer.

```
Arguments : pool — pool of domain elements
            exprs — pool of expressions
            vars — program variables
            fuzzingData — indexes provided by the fuzzer
1 Procedure testAssign
2     x̄ ⟵ pool[nextIndex(fuzzingData)]
3     v ⟵ vars[nextIndex(fuzzingData)]
4     e ⟵ exprs[nextIndex(fuzzingData)]
5     res̄ ⟵ assign(x̄, v, e)                    // execute assign transformer
6     assert(x̄ ≠ ⊥ ⇒ res̄ ≠ ⊥)                  // check property (2.1)
```

2.4 TESTING NUMERICAL DOMAINS

In this section, we provide the technical details of our approach for testing implementations of numerical domains.

2.4.1 *Test oracles*

Our test oracles are based on domain-independent mathematical properties of abstract operations. In the following, we give an overview of these properties and explain how they are checked by the test drivers.

Properties. Based on the abstract interpretation literature [51, 54] and earlier work on testing static analyzers [120], we identified 46 properties (Figure 2.3) that need to be satisfied by domain operations to ensure *soundness*, *precision*, and *termination*.

The *soundness* properties (marked with [S] in Figure 2.3) are required to ensure that an abstract-domain element over-approximates the concrete states it represents. We have discussed an example in Section 2.3.1 (property (2.1), i.e., 35).

To deal with the undecidability of most semantic program properties, static analyses over-approximate the set of concrete program behaviors and then infer or check properties on this abstraction. Since they are intended to compute an approximation, one cannot expect the operations of an abstract domain to be precise w.r.t. the concrete program execution. Therefore, our *precision* properties (marked with [P] in Figure 2.3) check that the domain operations do not lead to *unnecessary* information loss, that is, they compare the result of an operation to the most precise representable result as obtained by applying the best transformer (see Section 2.3). For instance, computing the join of the intervals $[1,1]$ and $[3,3]$ yields $[1,3]$, which loses the information that the variable is different from 2. Despite this inevitable information loss, a join operator should satisfy a number of precision properties (6, 10–13, 15); for example, property 13 prevents the join of $[1,1]$ and $[1,3]$ from returning $[0,4]$ or $\top$, which is sound, but unnecessarily imprecise.

	Partial order	
1	$\bot \sqsubseteq \overline{x}$	[P]
2	$\overline{x} \sqsubseteq \top$	[P]
3	$\overline{x} \sqsubseteq \overline{x}$	[P]
4	$\overline{x} \sqsubseteq \overline{y} \wedge \overline{y} \sqsubseteq \overline{z} \Rightarrow \overline{x} \sqsubseteq \overline{z}$	[P]
5	$\overline{x} \sqsubseteq \overline{y} \wedge \overline{y} \sqsubseteq \overline{x} \Rightarrow \overline{x} = \overline{y}$	[P]
	Join	
6	$\bot \sqcup \overline{x} = \overline{x}$	[P]
7	$\top \sqcup \overline{x} = \top$	[S]
8	$\overline{x} \sqsubseteq \overline{x} \sqcup \overline{y}$	[S]
9	$\overline{y} \sqsubseteq \overline{x} \sqcup \overline{y}$	[S]
10	$\overline{x} \sqcup \overline{y} = \overline{y} \sqcup \overline{x}$	[P]
11	$(\overline{x} \sqcup \overline{y}) \sqcup \overline{z} = \overline{x} \sqcup (\overline{y} \sqcup \overline{z})$	[P]
12	$\overline{x} \sqcup \overline{x} = \overline{x}$	[P]
13	$\overline{x} \sqsubseteq \overline{y} \Rightarrow \overline{x} \sqcup \overline{y} = \overline{y}$	[P]
14	$\overline{x} \sqcup \overline{y} = \overline{y} \Rightarrow \overline{x} \sqsubseteq \overline{y}$	[S]
15	$\overline{x} \sqcup (\overline{x} \sqcap \overline{y}) = \overline{x}$	[P]
	Meet	
16	$\bot \sqcap \overline{x} = \bot$	[P]
17	$\top \sqcap \overline{x} = \overline{x}$	[P]
18	$\overline{x} \sqcap \overline{y} \sqsubseteq \overline{x}$	[P]
19	$\overline{x} \sqcap \overline{y} \sqsubseteq \overline{y}$	[P]
20	$\overline{x} \sqcap \overline{y} = \overline{y} \sqcap \overline{x}$	[P]
21	$(\overline{x} \sqcap \overline{y}) \sqcap \overline{z} = \overline{x} \sqcap (\overline{y} \sqcap \overline{z})$	[P]
22	$\overline{x} \sqcap \overline{x} = \overline{x}$	[P]
23	$\overline{x} \sqsubseteq \overline{y} \Rightarrow \overline{x} \sqcap \overline{y} = \overline{x}$	[P]
24	$\overline{x} \sqcap \overline{y} = \overline{x} \Rightarrow \overline{x} \sqsubseteq \overline{y}$	[P]
25	$\overline{x} \sqcap (\overline{x} \sqcup \overline{y}) = \overline{x}$	[P]
26	$\mathit{disj}(\overline{x}, \overline{y}) \Rightarrow \overline{x} \sqcap \overline{y} = \bot$	[P]
	Join/Meet bounds	
27	$\forall \overline{b} : (\overline{x} \sqsubseteq \overline{b}) \wedge (\overline{y} \sqsubseteq \overline{b}) \Rightarrow (\overline{x} \sqcup \overline{y} \sqsubseteq \overline{b})$	[P]
28	$\forall \overline{b} : (\overline{b} \sqsubseteq \overline{x}) \wedge (\overline{b} \sqsubseteq \overline{y}) \Rightarrow (\overline{b} \sqsubseteq \overline{x} \sqcap \overline{y})$	[P]
	Widening	
29	$\overline{x} \sqsubseteq \overline{x} \nabla \overline{y}$	[S]
30	$\overline{y} \sqsubseteq \overline{x} \nabla \overline{y}$	[S]
31	$\overline{x} \nabla \bot = \overline{x}$	[P]
32	$\bot \nabla \overline{x} = \overline{x}$	[P]
33	∇ converges	[C]
	Assignment	
34	$\overline{x} \sqsubseteq \overline{y} \Rightarrow \mathit{assign}(\overline{x}, v, e) \sqsubseteq \mathit{assign}(\overline{y}, v, e)$	[P]
35	$\overline{x} \neq \bot \Rightarrow \mathit{assign}(\overline{x}, v, e) \neq \bot$	[S]
36	$\overline{x} = \bot \Rightarrow \mathit{assign}(\overline{x}, v, e) = \bot$	[P]
37	$\mathit{rep}(e, \overline{x}) \Rightarrow \mathit{assign}(\overline{x}, v, e) \neq \top$	[P]
	Projection	
38	$\mathit{assign}(\overline{x}, v, e) \sqsubseteq \mathit{project}(\overline{x}, v)$	[P]
	Conditional	
39	$\overline{x} \sqsubseteq \overline{y} \Rightarrow \mathit{cond}(\overline{x}, e) \sqsubseteq \mathit{cond}(\overline{y}, e)$	[P]
40	$\overline{x} = \bot \Rightarrow \mathit{cond}(\overline{x}, e) = \bot$	[P]
41	$\mathit{cond}(\overline{x}, e) \sqsubseteq \overline{x}$	[P]
	Narrowing	
42	$\overline{x} \sqcap \overline{y} \sqsubseteq \overline{x} \triangle \overline{y}$	[P]
43	$\overline{x} \triangle \overline{y} \sqsubseteq \overline{x}$	[P]
44	$\overline{x} \triangle \bot = \bot$	[P]
45	$\bot \triangle \overline{x} = \bot$	[P]
46	$\triangle$ converges	[C]

FIGURE 2.3: Algebraic properties of abstract domain operations. We classify them into soundness [S], precision [P], or convergence [C]. Note that all free variables are implicitly universally quantified and all variables refer to abstract-domain elements except for v and e, which refer to a program variable and an expression, respectively. The predicate $\mathit{disj}(\overline{x}, \overline{y})$ denotes that the intersection of the set of constraints from $\overline{x}$ and $\overline{y}$ is trivially empty, and the predicate $\mathit{rep}(e, \overline{x})$ expresses that e can be precisely represented in the abstract domain of $\overline{x}$.

The *convergence* properties (33 and 46, marked with [C] in Figure 2.3) require widening and narrowing to eventually reach a fixed point, which is necessary to ensure that the static analysis of loops and recursion terminates. Since convergence

Algorithm 2.3: Our test driver for checking whether the Octagons widening reaches a fixed point within k steps.

```
Arguments : pool — pool of domain elements
            k — maximum number of steps
            fuzzingData — indexes provided by the fuzzer
1  Procedure testOctagonsWidening
2      i ⟵ 0
3      x̄ ⟵ pool[nextIndex(fuzzingData)]
4      while true do
5          i ⟵ i + 1
6          ȳ ⟵ pool[nextIndex(fuzzingData)]
7          res̄ ⟵ x̄ ∇ ȳ
8          if res̄ = x̄ then
9              break                          // a fixed point is reached
10         x̄ ⟵ res̄
11         assert(i < k)
```

is a termination property, which cannot be tested, we instead check the stronger property that a fixed point is reached within a given number of iterations, as we explain later in this section.

The soundness and convergence properties need to hold for all abstract domains; some precision properties may not hold when the best transformers do not exist or cannot be computed [135]. For instance, domains based on incomplete lattices (such as Zonotopes) do not have a least upper bound for every pair of abstract elements. This can force $\sqcup$ to return a larger upper bound and, thus, violate property 27. For such domains, we require the developers to select the subset of precision properties that should be checked.

Executable oracles. We manually construct a test driver for every property in Figure 2.3. This driver selects suitable inputs for each of the free variables of the tested property, evaluates it, and checks that it holds. For property 3, it selects a domain element $\bar{x}$ from the pool and checks that $\sqsubseteq$ yields `true` when applied to $\bar{x}$ and $\bar{x}$.

Translating properties into executable oracles is straightforward for most soundness and precision properties, but slightly more involved for convergence properties. Algorithm 2.3 shows our test driver for checking whether the Octagons widening converges after k iterations [121]. The driver computes an increasing chain of abstract elements $\bar{x}$ (as checked by property 29), each obtained by widening the previous abstract element with an arbitrary abstract element $\bar{y}$. Widening converges if the $\bar{x}$-chain becomes stable, here, within k steps. We observed that $k = 100$ is a sufficient upper bound for all our tested analyzers because widening converges much faster in practice. Note that, for most abstract domains (such as Intervals, Octagons, etc.), widening can be applied to arbitrary elements. There are, however, some exceptions; for instance, the Polyhedra widening requires mono-

tonicity of its operands (that is, $\overline{x} \sqsubseteq \overline{y}$). In such cases, we use a slightly different test driver, which replaces $\overline{x} \nabla \overline{y}$ from Algorithm 2.3, line 7 with $\overline{x} \nabla (\overline{x} \sqcup \overline{y})$, because $\overline{x} \sqsubseteq \overline{x} \sqcup \overline{y}$ (see property 8). As a consequence of this monotonicity precondition, property 31 needs to hold for the Polyhedra domain only for $\overline{x} = \bot$.

2.4.2 *Input data*

Testing the properties from Figure 2.3 requires three kinds of input data: (1) program variables, (2) expressions over them (for the *assign* and *cond* transformers), and (3) abstract-domain elements that contain constraints over these variables. We construct this data as follows.

Program variables. All our test cases operate on a set of predefined integer variables. The number of variables must be sufficiently small to keep the memory consumption and execution time of the test cases low. On the other hand, abstract-domain optimizations, such as decomposition, require enough variables to obtain nontrivial partitions [148]. In our implementation, the number of variables is configurable; we use eight variables in our experiments.

Expressions. Testing assignments requires expressions. For the numerical domains considered in this work, these expressions are linear sums over the program variables with integer coefficients, which are chosen by the fuzzer to increase the likelihood of constructing suitable expressions. Note that, to test precision properties, we also need to generate expressions that are not representable in the domain under test (for instance, polyhedral constraints for testing intervals). For the *cond* transformer, we obtain boolean expressions by comparing the linear sums to zero.

Domain elements. As explained in Section 2.3, we create a pool of abstract-domain elements (such as intervals or octagons) in two steps: step 1 populates the pool by constructing elements using a combination of boundary and random testing, and step 2 expands it by applying domain operations to existing pool elements.

Besides $\top$ and $\bot$, step 1 also creates simple domain elements that contain only one constraint on the predefined program variables. More complex domain elements are constructed in step 2. Table 2.1 shows the structure of simple domain elements for widely-used numerical domains. These elements can be constructed by choosing values for the constants k (e.g., the bounds of an interval) and the coefficients c. By default, we pick them from a small set of *predefined values* (e.g., boundary values such as 0 or ∞) and a small set of *arbitrary values*. The use of predefined values is optional and their set is configurable (see Section 2.5).

These values depend on both the domain under test and its implementation. For instance, octagonal coefficients must be in $\{-1, 0, 1\}$, while polyhedral coefficients can be arbitrary integers. Moreover, different implementations represent numbers differently; e.g., we use the predefined values $\{\mathit{INT\_MIN}, 0, \mathit{INT\_MAX}\}$ for integer intervals if the implementation uses machine integers, $\{-\infty, 0, \infty\}$ for arbitrary-precision integers, and additionally *NaN* for floating-point representations. Even

Domain	Abstract element with one constraint
Intervals	$\{k_l \leq v_i \leq k_u\}$
Zonotopes	$\{v_i = \frac{k_l+k_u}{2} + \frac{k_u-k_l}{2} * \varepsilon_i\}$
Octagons	$\{c_i v_i + c_j v_j \odot k\}$
Polyhedra	$\{c_0 v_0 + c_1 v_1 + \ldots + c_{dim-1} v_{dim-1} \odot k\}$

v: variables; c: coefficients; $\varepsilon \in [-1, 1]$; k: constants; $\odot \in \{\geq, =\}$

TABLE 2.1: Abstract elements with one constraint from widely-used numerical domains.

though we focus on integer program variables, our experiments show that internal floating-point computations may still lead to rounding errors (see Section 2.5).

The values in both sets are *not* chosen by the fuzzer. However, the fuzzer can still control the pool of domain elements by selecting suitable operations in step 2. This step constructs more diverse domain elements, usually with more complex constraints, by applying domain operations to the existing elements. Step 2 makes use of all domain operators that yield domain elements ($\sqcup$, $\sqcap$, ∇), as well as the abstract transformers *assign* and *project*. We omit narrowing, which is not supported by all domains, and conditionals, which are already covered (conditionals are implemented by intersecting a domain element with a linear constraint, i.e., in the same way in which we construct the initial elements from Table 2.1). Using all these domain operations not only allows us to detect errors in their implementation (such as the missing closure in assignments that we discussed in Section 2.3), but it also efficiently generates a diverse set of *valid* domain elements. While it is theoretically possible to create a new element by generating an arbitrary set of constraints, such an approach would often produce unsatisfiable conjunctions of constraints, represented by the already considered $\bot$ element.

2.5 EVALUATION

To evaluate the effectiveness of our technique, we used it to test two complex libraries for numerical analysis, namely APRON [90] and ELINA [148]. We were able to find errors in several of the implemented domains, all of which are confirmed and most are also fixed. APRON is a mature library with $\approx$ 73 000 LOC, used in many academic and industrial static analyzers, such as Astrée [30] and PAGAI [85], as well as in the CPAchecker verification platform [24]. ELINA is a more recent library with $\approx$ 60 000 LOC, which uses highly-optimized algorithms based on online decomposition to achieve significant speedups [147, 148]; these algorithms are difficult to implement correctly.

In our experiments, we considered three variants of APRON with different internal representations for numerical values (Section 2.5.1 and Section 2.5.2), and two

	Maximum execution time (s)	
Domain	**APRON**	**ELINA**
Intervals	750	not considered
Zonotopes	2 400	not considered
Octagons	1 900	700
Polyhedra	17 700	1 800

TABLE 2.2: Maximum execution time per test driver.

versions of ELINA (Section 2.5.3). We also evaluated different configurations of our technique (Section 2.5.4), and compared it to pure fuzzing and DSE (Section 2.5.5).

Experimental setup. Since the tested domains have different complexity (i.e., the implementation of Polyhedra is significantly slower), we estimated the maximum execution time required to test each property for approximately 1 million times for Intervals, Zonotopes, and Octagons and half a million times for Polyhedra (see Table 2.2). The values are smaller for ELINA than for APRON because ELINA's code is highly optimized. Intervals and Zonotopes were not considered for ELINA, as they were not part of the tested artifacts. We used LibFuzzer [113] (version 5.0.0) to construct the input data. All the experiments were performed on a 3.3 GHz Intel Xeon E5-4627 v2 CPU with 236 GB memory and RAID6 HDD.

2.5.1 *APRON Double and Rll*

APRON supports different internal representations for numerical values. For instance, the *Double* representation uses floating-point numbers, while *Rll* uses an approximation of rational numbers based on two 64-bit integers for the numerator and denominator. Compared to APRON MPQ, which uses arbitrary-precision rationals (see Section 2.5.2), these representations offer better performance, but may lose precision and cause non-termination [169]. Intervals and Octagons support Double, while Rll is available for Polyhedra.

Our experiments indeed uncovered soundness, precision, and termination problems in several domains from APRON (0.9.10, the latest version in 2018), as shown in Table 2.3. Here, the three versions of Rll refer to different test configurations that we discuss later; the domain implementation is always the same. The third column presents the total number of properties from Figure 2.3 that we attempted to test for each domain. We tested only the first 41 properties for Intervals and Polyhedra since narrowing is not implemented for Intervals in APRON and not mathematically defined for Polyhedra. The reported violations in the fourth column are not necessarily all caused by different bugs. Nevertheless, observing multiple violations caused by the same bug can provide additional information for error localization. We used a configuration that initializes the pool with 32 elements (step 1 of the pool population), includes *LONG_MIN* and *LONG_MAX* as boundary values, and applies 16 operations to generate more complex domain elements (step 2).

Variant	Domain	#Tests	#Failed tests			Causes
Double	Intervals	41	0 [S]	0 [P]	0 [E]	–
Double	Octagons	46	0 [S]	3 [P]	0 [E]	rounding
Rll v1	Polyhedra	41	0 [S]	0 [P]	41 [E]	overflow
Rll v2	Polyhedra	41	5 [S]	15 [P]	21 [E]	overflow
Rll v3	Polyhedra	41	5 [S]	19 [P]	4 [E]	overflow

[S] = soundness; [P] = precision; [E] = error (causing the driver to crash/time out)

TABLE 2.3: Results for APRON Double and Rll. The third column reports how many of the properties from Figure 2.3 were applicable for each tested domain.

This is configuration C2 from Table 2.6, which we discuss in Section 2.5.4, together with measurements on the testing time.

Intervals and Octagons. All our generic properties hold for Intervals, the simplest domain we tested. Nonetheless, we indirectly found imprecisions for $\sqsubseteq$ and $\sqcup$, by testing APRON's implementation of Zonotopes [79] (discussed in Section 2.5.2), which uses the Interval operations. These issues were confirmed and fixed.

For Octagons, three precision properties (17, 22, and 23) are violated, because the equality test gives imprecise results due to rounding errors. In Figure 2.4, we show an example input generated by our approach that violates property 17. *oct* represents a call to the Octagons constructor, which intersects the provided constraint with $\top$. The root cause of the imprecision is the underlying double representation. For oct_3, *LONG_MAX* cannot be precisely represented as a double value. The resulting rounding error gives approximate results in the subsequent computations and makes the assertion fail. Note that we do not define separate properties for equality in Figure 2.3, because for most of the domains this check is implemented as a double inclusion: $\overline{x} = \overline{y} \Leftrightarrow \overline{x} \sqsubseteq \overline{y} \wedge \overline{y} \sqsubseteq \overline{x}$. The Octagons are an exception, as they define the equality based on closure [121].

Polyhedra. With the same test configuration (C2), all the tests fail to terminate for Polyhedra Rll (Rll v1 in Table 2.3). The problem is that APRON enters infinite loops during step 1 of the pool construction because of unhandled arithmetic overflows. The bug can be seen when constructing at least two consecutive domain elements, as in Figure 2.5. The second constructor call (*poly*) enters an infinite loop.

$$
\begin{aligned}
&oct_1 \longleftarrow oct(-x_0 - x_5 + 1 \geq 0)\\
&oct_2 \longleftarrow assign(oct_1, x_2, LONG\_MIN)\\
&oct_3 \longleftarrow oct(x_0 + LONG\_MAX \geq 0)\\
&oct_4 \longleftarrow oct_3 \sqcap oct_2\\
&assert(\top \sqcap oct_4 = oct_4)
\end{aligned}
$$

FIGURE 2.4: Input violating property 17 for APRON Octagons.

$poly_1 \longleftarrow poly(-x_5 - x_6 + x_7 \geq LONG\_MAX)$
$poly_2 \longleftarrow poly(-x_4 - x_5 - x_6 - x_7 \geq LONG\_MIN)$

FIGURE 2.5: Input entering an infinite loop for APRON Polyhedra.

This bug causes all test drivers to time out before they even reach the test oracle. To work around this issue and look for additional bugs, we replaced *LONG_MIN* and *LONG_MAX* by *INT_MIN* and *INT_MAX* as predefined values. In this case (Rll v2 in Table 2.3), step 1 of the pool construction succeeds, but for 21 test drivers, step 2 times out. The remaining 20 test drivers lead to violations of soundness and precision properties. The root cause of all these failures is unhandled arithmetic overflows in various operators. Dropping predefined values entirely (Rll v3 in Table 2.3) allows us to construct the pool in all but four cases. In total, 24 soundness and precision properties fail, due to overflow.

2.5.2 *APRON MPQ*

MPQ is an APRON variant that uses arbitrary-precision rationals for its internal representation. For sub-polyhedral domains (i.e., Intervals, Octagons, and Polyhedra) this variant is supposed to be sound and precise [169]. Our experiments partially confirm these theoretical guarantees: with the same setup as for APRON Double and Rll, we did not find any (generic) property violations in APRON MPQ for the three sub-polyhedral domains. However, we revealed imprecisions for Intervals indirectly, by testing Zonotopes. Moreover, to further validate our technique, we asked three experts in abstract interpretation to insert bugs in any of the sub-polyhedral domains. Our results are presented in the following paragraphs.

Zonotopes. As opposed to sub-polyhedral domains, the structure of Zonotopes is an incomplete lattice. For this reason, not all the precision properties from Figure 2.3 are expected to hold (e.g., as explained in Section 2.4.1, the least upper bound may not exist for every pair of elements). Moreover, join creates new, input-related constants [79] and thus the operator is by design non-commutative.

Initially, the pool construction step did not succeed for any of the tests, due to a memory bug in the meet operator. After this issue was fixed, we detected additional memory exceptions, raised when creating a high number of input-related constants. Our tests also revealed imprecisions in the implementation of the equality check, meet and project operations. Moreover, we discovered a precision bug in the partial order. Soundness property 8 uses $\sqsubseteq$ to check if the result of a join over-approximates its operands, and the bug led to a violation of this property. The developers concluded that the root cause is an imprecision in the implementation of the Intervals domain, when one of the operands of $\sqsubseteq$, $\sqcup$, or ∇ is $\bot$, represented in its canonical form through the empty interval $[1, -1]$. If $\bot$ is not handled as a special case, $[1, -1] \sqcup [-10, -5] = [-10, -1]$, for example, instead of $[-10, -5]$. This imprecision is independent of the internal representation used

Domain	#Seeded bugs	#Found bugs	#Failed tests		
Intervals	5	4	1 [S]	14 [P]	0 [E]
Octagons	6	5	2 [S]	15 [P]	5 [E]
Polyhedra	6	5	4 [S]	10 [P]	52 [E]

[S] = soundness; [P] = precision; [E] = error (causing the driver to crash/time out)

TABLE 2.4: Results for APRON MPQ with seeded bugs.

for numbers. Our tests for Intervals could not detect it directly, because our properties are generic and do not check the precision based on the Intervals-specific definitions. All these issues were fixed by the APRON developers.

Seeded bugs in sub-polyhedral domains. We also asked three abstract interpretation researchers, a post-doc and two senior PhD students with a broad experience in implementing and using various types of abstract domains and static analyses, to seed semantic bugs for our evaluation. Each expert had the task of inserting at least five soundness or precision bugs (at least one of each type) in any of the sub-polyhedral domains. They could seed them directly in the functions performing $\sqsubseteq$, $\sqcup$, $\sqcap$, ∇, $\triangle$ (if defined), *assign*, *project*, or *cond* or in any other function reachable from them. We believe that the seeded bugs are representative for the kind of semantic errors that occur during the development of abstract domains.

The cumulative results are summarized in Table 2.4. For each domain, we show how many bugs were seeded, how many our work found, and whether we observed the bugs through violations of soundness or precision properties, or through crashes and assertion failures in APRON's internal consistency checks. In total, we were able to find 14 out of the 17 seeded bugs. In the following, we present two bugs that we detected and explain why the other three could not be identified.

(1) A seeded bug in Intervals is caused by a slightly modified version of an unsound definition for the widening operator [115]. This definition uses $\leq$ instead of $>$ to compare two bounds, which leads to a violation of the soundness property 30. Our tests reveal this problem. (2) A seeded bug in Octagons removes the closure in one special case of the assignment transformer. As explained in Section 2.3, we detect this bug by generating an assignment during step 2, followed by a meet, which violates four of our precision properties, as the equality becomes imprecise.

While our approach found the vast majority of the seeded bugs, there were three it did not detect. (1) One seeded bug makes the Octagons closure less precise. Detecting this problem would require additional, octagon-specific precision properties for closure [121]. This can be easily done, but our focus here is on domain-independent properties. (2) Another seeded bug affects the precision of widening for Intervals in a way that does not violate the properties from Figure 2.3. Detecting it would again require additional, domain-specific properties. (3) The last undetected bug changes the assignment operator for Polyhedra to act like *project*, making it trivially sound, but imprecise. This bug can be easily found if the inputs are polyhedra of dimension 1 (i.e., intervals). Our default configuration excludes

Variant	#Tests	#Failed tests			Causes
EP1	41	4 [S]	10 [P]	27 [E]	overflow, ∇
EOD	41	0 [S]	3 [P]	0 [E]	rounding
EO	41	0 [S]	3 [P]	0 [E]	rounding
EP2	41	0 [S]	0 [P]	41 [E]	overflow, assertions
EP3	41	4 [S]	18 [P]	19 [E]	overflow, ∇

[S] = soundness; [P] = precision; [E] = error (causing the driver to crash)

TABLE 2.5: Results for different variants of ELINA.

such elements; adjusting the range of dimensions to include the value 1 leads to a violation of the property 37, revealing the imprecision.

2.5.3 *ELINA*

To evaluate how effective our technique is on highly optimized implementations, we applied it to test ELINA. We used the same configuration as for APRON (C2) on the following abstract domains: *EP1* – Polyhedra with decomposition [148], *EOD* – Octagons with decomposition [149], *EO* – Octagons without decomposition [149], *EP2* – more recent version (including bug fixes) of *EP1*, *EP3* – decomposed Polyhedra with further optimizations [147]. *EP1* is the implementation from the artifact [67] from POPL'17 [148], and the other variants are part of the artifact [68] from POPL'18 [147]. Note that all ELINA domains are based on floating-point numbers (not on the slower arbitrary-precision rationals). This design decision may compromise precision to achieve high performance.

Initially, the pool construction for *EP1* failed for all the tests due to corner cases like $\frac{LONG\_MIN}{-1}$. This step was also not successful for *EP2* and *EP3* if the polyhedra had *LONG_MAX* coefficients (a similar issue as for APRON Double, see Section 2.5.1). To find additional errors, we limited the set of predefined values to $\{INT\_MIN, -1, 0, 1, INT\_MAX\}$. Our results are summarized in Table 2.5. Manual inspection of the failed tests revealed that most of them were caused by arithmetic overflows in different operations (e.g., *assign* and $\sqcap$). We also found issues due to overly restrictive assertions in the code, as well as an incorrect implementation of widening for certain cases (e.g., widening with $\bot$). We reported these issues (and others), and they were fixed by the developer.

One of the most interesting bugs we found in *EP1* is related to an inconsistency between the two polyhedral representations, i.e., constraints and generators. To improve performance, ELINA uses an optimized implementation of the Chernikova algorithm [175] for incremental conversion and applies all the operators on decomposed polyhedra. Such optimizations make it much more difficult to keep the two representations in sync. Our pool-construction approach created polyhedra with inconsistent internal states, by applying sequences of meet and join operations that use different internal representations (meet is performed on the constraints repre-

Configuration	Initial pool size	#Operations	Predefined values?
C1	2	16	yes
C2	32	16	yes
C3	1 024	16	yes
C4	32	0	yes
C5	32	64	yes
C6	32	16	no

TABLE 2.6: Different configurations of our technique.

sentation, while join is efficiently implemented using generators [148]). As a result, the subsequent test for the soundness of *assign* failed (property 35), because the transformer returned $\bot$. The same bug was reported by another ELINA user [180] a few days before we reported it, which shows that our approach detects bugs that are relevant for users of numerical libraries. It was fixed in the meantime.

Since narrowing was not available for Octagons through ELINA's APRON interface (which we use for testing), we did not include the corresponding test drivers in this experiment (only the first 41 properties were tested, as shown in Table 2.5). For both variants of Octagons, like for APRON Double (see Section 2.5.1), the properties 17, 22, and 23 were violated. These imprecisions are caused by rounding errors when performing closure and, implicitly, in the equality tests. We reported the issues and they were confirmed. As obtaining precise results with finite-precision representations is very challenging, these problems are not yet fixed.

2.5.4 *Sensitivity analysis*

Our technique relies on three main configurable parameters: the size of the initial pool from step 1 (parameter `size` in Algorithm 2.1), the number of operations applied in step 2 (parameter `nbops` in Algorithm 2.1), and whether predefined values are used to construct elements and expressions (parameter `predefined` in Algorithm 2.1). We now assess the impact of these parameters on its effectiveness.

For this experiment, we used the three versions of APRON MPQ with the bugs seeded by the experts and the configurations shown in Table 2.6. The results are presented in Table 2.7, Table 2.8, and Table 2.9. All three tables have the same structure: for each seeded bug, we report the abstract domain in which it was inserted, the violated properties that exposed it, and the execution time until each configuration detects the violation or the implementation throws an error (which we also consider as a detected violation, since without the seeded bug, the corresponding domain operation should neither crash, nor fail internal consistency checks). As shown in the tables, configurations C2, C5, and C6 all find the maximum number of violations, and implicitly all the bugs, within the time limits defined in Table 2.2. However, C2 does so significantly faster than C5 and slightly faster than C6.

Domain	Failed test	C1	C2	C3	C4	C5	C6
Intervals	15 [P]	-	0.3	1.36	0.1	1.48	0.38
Intervals	17 [P]	-	0.05	1.78	0.05	0.05	0.05
Intervals	18 [P]	-	0.51	1.82	0.53	0.6	0.53
Intervals	19 [P]	-	0.44	1.89	0.65	0.6	0.52
Intervals	23 [P]	-	2.6	1.34	0.93	3.06	0.99
Intervals	24 [P]	-	1.94	-	-	17.29	2.49
Intervals	25 [P]	-	0.33	1.34	0.1	1.71	0.4
Intervals	30 [S]	-	0.36	1.25	0.13	0.58	0.52
Octagons	6 [P]	0.04	0.05	0.49	0.1	0.06	0.06
Octagons	9 [S]	0.34	4.73	-	0.18	30.7	3.97
Octagons	10 [P]	0.95	3.05	-	0.13	2.81	5.12
Octagons	11 [P]	6.25	8.35	-	0.5	294.26	2.59
Octagons	13 [P]	0.45	3.23	-	0.22	108.69	7.67
#Failed tests (max 13)		**5**	**13**	**8**	**12**	**13**	**13**
#Bugs found (out of 3)		**1**	**3**	**3**	**3**	**3**	**3**

TABLE 2.7: Impact of different configurations on finding the bugs seeded by Expert 1. The last six columns show the time (s) needed by each configuration from Table 2.6. We grouped the failed tests by the seeded bugs they reveal (dashed lines).

Initial pool size. When the initial poll includes just $\top$ and $\bot$ (as in C1), no violations are detected for Intervals in comparison to C2, that is, 4 bugs are missed. On the other hand, a very large initial pool (C3) increases the execution time without detecting all violations. We attribute this to the fact that the size of the initial pool directly influences the number of possible arguments to the subsequent operations in step 2 and to the test oracle; exploring all of them takes more time without necessarily being more effective.

Number of operations. Without using step 2 of the pool construction (C4), some bugs and property violations are missed (see Section 2.3 for an example). However, for our technique to be effective, the number of operations should not be too large since it increases the execution time. C5 is usually slower than C2 even though both configurations find the same number of violations.

Predefined values. For APRON MPQ, which uses arbitrary precision rationals, both C2 and C6 find all the violations, C2 being slightly faster. Predefined values are particularly important for testing abstract-domain implementations based on fixed-precision numbers such as APRON Double and ELINA; as our experiments show (see Section 2.5.1 and Section 2.5.3), these implementations may suffer from arithmetic overflows and rounding errors.

Domain	Failed test	C1	C2	C3	C4	C5	C6
Intervals	30 [S]	-	0.72	1.24	0.27	2.1	0.45
Octagons	15 [P]	0.5	error	109.56	error	error	3.5
Octagons	16 [P]	0.06	0.12	2.47	0.06	0.13	0.05
Octagons	17 [P]	0.72	2.72	-	-	error	error
Octagons	18 [P]	error	error	23.2	error	error	error
Octagons	19 [P]	error	error	-	error	error	error
Octagons	20 [P]	0.8	error	771.25	error	error	error
Octagons	21 [P]	error	1.09	7.49	error	1.41	0.9
Octagons	22 [P]	0.72	error	-	error	error	error
Octagons	23 [P]	0.73	error	27.81	error	error	3.65
Octagons	24 [P]	error	error	-	error	error	error
Octagons	25 [P]	0.56	0.87	4.39	error	error	1.03
Octagons	28 [P]	error	error	-	error	error	error
Octagons	42 [P]	0.93	5.17	112.34	error	error	1.5
Octagons	38 [P]	0.05	0.07	2.04	0.05	0.07	0.14
Polyhedra	6 [P]	0.05	0.12	63.4	0.11	0.23	0.18
Polyhedra	9 [S]	1.22	30.03	-	0.21	44.47	312.41
Polyhedra	10 [P]	0.65	2.05	-	0.35	295.41	42.94
Polyhedra	11 [P]	54.56	419.51	-	3.06	>4 h	1487.56
Polyhedra	13 [P]	1.18	106.69	-	0.44	483.82	86.3
Polyhedra	34 [P]	5.46	45.58	-	0.62	33.76	85.97
Polyhedra	36 [P]	0.07	0.14	60.4	0.12	0.22	0.17
Polyhedra	37 [P]	0.48	13.0	-	0.16	4.87	217.27
Polyhedra	38 [P]	0.46	41.75	-	0.73	5.42	449.95
#Failed tests (max 24)		**23**	**24**	**12**	**23**	**24**	**24**
#Bugs found (out of 5)		**4**	**5**	**5**	**5**	**5**	**5**

TABLE 2.8: Impact of different configurations on finding the bugs seeded by Expert 2. We excluded two bugs seeded in Polyhedra that caused assertions failures in APRON for all configurations, and masked the other seed bugs. The columns have the same structure as in Table 2.7.

2.5.5 *Fuzzing and dynamic symbolic execution*

In this section, we compare our technique to pure grey-box fuzzing and DSE. In particular, we use LibFuzzer [113] (version 5.0.0, i.e., the same version as for our tool) and KLEE [41] (version 1.4.0), a state-of-the-art DSE engine, to generate inputs for the test oracles corresponding to the properties from Figure 2.3. For a fair comparison, we wrote *alternative test drivers* that do not create and populate a pool of abstract elements. Instead, they allow LibFuzzer and KLEE to directly generate

Domain	Failed test	C1	C2	C3	C4	C5	C6
Intervals	5 [P]	-	2.09	15.32	9.34	2.39	787
Intervals	13 [P]	-	0.86	3.51	0.47	4.94	1.0
Intervals	23 [P]	-	0.62	2.19	0.51	1.49	1.17
Intervals	28 [P]	-	269	3.59	0.12	1.0	2.02
Intervals	34 [P]	-	12.54	70.86	1.23	11.34	12.21
Intervals	39 [P]	-	4.37	25.21	1.9	5.99	24.99
Octagons	9 [S]	0.67	0.83	3.53	0.11	0.27	0.4
Octagons	10 [P]	0.45	0.23	2.48	0.11	0.88	0.87
Octagons	13 [P]	0.65	0.55	24.58	0.32	28.67	23.94
Octagons	17 [P]	4.32	23.61	-	-	117.56	16.37
Octagons	22 [P]	9.78	24.22	-	-	17.3	11.25
Octagons	23 [P]	4.26	9.3	-	-	15.08	11.05
Octagons	25 [P]	3.78	11.09	-	-	81.92	9.17
Polyhedra	7 [S]	0.07	error	error	2.02	error	error
Polyhedra	8 [S]	0.07	0.18	64.47	0.16	0.13	error
Polyhedra	9 [S]	0.09	0.23	66.55	0.17	0.13	error
Polyhedra	11 [P]	error	error	error	1.0	error	error
Polyhedra	12 [P]	0.08	0.19	65.96	0.29	0.18	0.2
Polyhedra	13 [P]	0.08	0.18	64.67	0.19	0.22	0.2
Polyhedra	25 [P]	0.08	0.2	61.76	0.19	0.14	0.19
Polyhedra	27 [P]	2.99	0.21	65.71	0.26	0.15	error
#Failed tests (max 21)		**15**	**21**	**17**	**17**	**21**	**21**
#Bugs found (out of 4)		**3**	**4**	**3**	**3**	**4**	**4**

TABLE 2.9: Impact of different configurations on finding the bugs seeded by Expert 3. The columns have the same structure as in Table 2.7.

the coefficients and constants of up to 50 constraints per element. The test oracles are the same as in our test drivers.

Grey-box fuzzing. We ran LibFuzzer (with default options) on the three APRON MPQ versions with seeded bugs, using the same time limits as in Table 2.2, and compared the results with our configuration C2. LibFuzzer detected 45 out of the 58 property violations that we found. It revealed these violations generally faster than our approach for Intervals and Polyhedra, but slower for Octagons.

A manual inspection of the generated counterexamples shows that our technique produces significantly simpler and more readable test inputs, which was very useful in debugging the detected issues. For instance, the following is an example octagon generated by LibFuzzer when directly choosing the constraints: $\overline{x} = \{-x_0 + x_1 = 9999 \wedge x_0 + x_1 = 19998 \wedge -x_0 + x_5 = 19998 \wedge x_0 + x_5 = 29997 \wedge -x_1 + x_5 = 9999 \wedge x_1 + x_5 = 39996 \wedge -x_0 + x_6 = 0 \wedge x_0 + x_6 = 9999 \wedge x_1 + x_6 = 19998 \wedge x_5 + x_6 = 29997 \wedge x_1 - x_6 = 9999 \wedge x_5 - x_6 = 19998 \wedge x_1 + x_7 = 0 \wedge x_5 +$

Domain	Failed test	Our work	Fuzzing
Intervals	5 [P]	2.09	0.39
Intervals	13 [P]	0.86	0.69
Intervals	23 [P]	0.62	0.49
Intervals	28 [P]	0.27	0.71
Intervals	34 [P]	12.54	0.39
Intervals	39 [P]	4.37	31.07
Octagons	9 [S]	0.83	7.49
Octagons	10 [P]	0.23	8.01
Octagons	13 [P]	0.55	15.16
Octagons	17 [P]	23.61	-
Octagons	22 [P]	24.22	-
Octagons	23 [P]	9.3	-
Octagons	25 [P]	11.09	-
Polyhedra	7 [S]	error	-
Polyhedra	8 [S]	0.18	-
Polyhedra	9 [S]	0.23	-
Polyhedra	11 [P]	error	-
Polyhedra	12 [P]	0.19	0.07
Polyhedra	13 [P]	0.18	-
Polyhedra	25 [P]	0.2	-
Polyhedra	27 [P]	0.21	-
#Failed tests (max 21)		**21**	**10**
#Bugs found (out of 4)		**4**	**3**

TABLE 2.10: Comparison of our technique with fuzzing for the bugs seeded by Expert 3.

$x_7 = 9999 \wedge -x_0 - x_7 = 9999 \wedge x_0 - x_7 = 19998 \wedge x_1 - x_7 = 29997 \wedge x_5 - x_7 = 39996 \wedge -x_6 - x_7 = 9999 \wedge x_6 - x_7 = 19998\}$.

For the same violated property (number 12), our approach generated the octagon $\overline{x} = \{-x_0 - x_5 + 1 \geq 0 \wedge x_7 = LONG\_MIN\}$, which was obtained by performing the assignment $x_7 = LONG\_MIN$ on the octagon from the initial pool $\{-x_0 - x_5 + 1 \geq 0\}$.

Table 2.10 provides additional details for the bugs seeded by Expert 3. The last two columns show the time it takes to detect a property violation for our technique (with C2) and for LibFuzzer, respectively. LibFuzzer missed one of the Octagon bugs. Moreover, its results for Polyhedra suggest that the implementation of join is imprecise since only property 12 is violated. In contrast, our work revealed that the expert seeded a more serious soundness bug (properties 7–9 are also violated).

Dynamic symbolic execution. As KLEE does not try to explore all execution paths in external libraries and APRON makes heavy use of libraries, we performed the comparison on the *EOD* and *EO* ELINA domains, using KLEE's default options.

Our alternative test drivers use ELINA's functions directly, not its APRON interface, to avoid external library calls. However, all but 1 test throw an error for both domains, since KLEE is not able to model malloc instructions with symbolic sizes [96]. To overcome this limitation, we extended it with an option for specifying the upper bound for the symbolic size; we used 8 192 in our experiments. For each tested domain, our technique detected 3 violated properties (see Table 2.5), but KLEE was not able to detect any, even with a time limit of 17 700 s (the 25-fold of the time limit used for our work). We believe this is due to the fact that ELINA heavily relies on floating-point arithmetic, which KLEE does not handle very well.

2.5.6 *Threats to validity*

We identified two threats to the validity of our experiments:

Test generation tool. Our comparisons to the alternative approaches focus on one fuzzer and one DSE tool. Since we chose mature, state-of-the-art tools, we believe that our results are representative. Similarly, we did not use alternative grey-box fuzzers when evaluating our own approach, but we did use the same fuzzer (LibFuzzer) both for generating the inputs for our technique and as a related work. Since most fuzzers make no assumptions about the code under test and support a uniform input format (an array of bytes or a file), we do not expect to see significantly different results for other fuzzers.

Random initialization. Our pool initialization step chooses some of the coefficients and constants randomly. To ensure that our results are deterministic, all test drivers use the same, predefined random seed.

2.6 TESTING LEARNING-BASED ABSTRACT INTERPRETATION TRANSFORMERS

Our technique is not limited to the classical domain operations. It can be applied also to learning-based abstract interpretation transformers (LAIT); these were recently introduced by He et al. [83] and rely on machine learning algorithms to improve the performance of the analysis. The key idea of this work is that sequences of domain elements generated by the static analyzer can contain a high number of redundant constraints, which are not necessary for computing the final invariants. Thus He et al. learn a neural classifier that identifies and removes these constraints. The approach was instantiated for synthesizing *approximate join transformers* for Octagons and Polyhedra, which are expected to be sound and almost as precise as the traditional ones [83]. The implementation is integrated into ELINA [69] and combines Python libraries for machine learning with ELINA's highly optimized C code. Identifying errors in the actual implementation of LAIT requires techniques (like ours) that can effectively explore complex libraries and heterogeneous code.

For this experiment, we considered the commit [71], not the PLDI'20 artifact [84], since it provides a more similar interface to the classical join transformers, which is more suited for unit testing (the artifact focuses on generating loop invariants

Variant	#Tests	#Failed tests			Causes
EP-LAIT1	10	0 [S]	1 [P]	9 [E]	$\perp$ elements or loop head
EO-LAIT1	10	0 [S]	0 [P]	9 [E]	$\perp$ elements or loop head
EP-LAIT2	10	0 [S]	6 [P]	4 [E]	overflow, *project*
EO-LAIT2	10	0 [S]	0 [P]	0 [E]	-

[S] = soundness; [P] = precision; [E] = error (causing the driver to crash)

TABLE 2.11: Results for different variants of ELINA LAIT.

because this is the precision metric used in their evaluation [83]). This code version corresponds to the variants EP-LAIT1 (for Polyhedra) and EO-LAIT1 (for Octagons) from Table 2.11. As LAIT is not available through the APRON interface (used in Section 2.5.3), we wrote ELINA-specific test drivers.

Recall from Figure 2.3 that join is applied to two abstract elements, $\overline{x}$ and $\overline{y}$. However, the corresponding LAIT functions (designed for computing loop invariants) accept as inputs two additional parameters: an abstract element representing the loop head and the result of the precise join of $\overline{x}$ and $\overline{y}$. In our test drivers, the loop head is also selected by the fuzzer from the pool. Nevertheless, we did not replace the domain operations used to increase the pool with the LAIT ones, since the approximate meet, widening, etc. are not implemented in ELINA. This approach allows us to clearly separate the usage of ELINA transformers and LAIT and facilitates error localization (we illustrate this aspect later in this section).

We tested all the 10 properties applicable for join (6–15 from Figure 2.3). We used configuration C2 (see Table 2.6) and the predefined values $\{LONG\_MIN, -1, 0, 1, LONG\_MAX\}$ for Octagons and $\{INT\_MIN, -1, 0, 1, INT\_MAX\}$ for Polyhedra, to avoid infinite loops during the pool construction (as described in Section 2.5.3). The results are presented in Table 2.11, which has the same structure as Table 2.5.

Initially (Table 2.11, rows 1–2), all Polyhedra tests and 9 Octagons tests failed, most of them causing the test driver to crash in the LAIT functions that compute the partitions. This was due to corner cases in which at least one input element or the loop head is $\perp$. We reported these errors and they were fixed. The developers confirmed that such test cases were not included in their dataset, so our approach could be used also for generating additional benchmarks for training or testing the underlying machine learning algorithm.

After applying the provided fixes [70], we retested the domains (these modified versions are the variants EP-LAIT2 and EO-LAIT2 from Table 2.11). For Polyhedra, the properties 6, 10–13, and 15 were violated, the approximate join transformer being imprecise mostly due to overflow. However, in four cases, step 2 of our pool construction approach did not succeed, revealing an error in ELINA's code for handling *project*; the bug can be reproduced without calling LAIT, by using only sequences of *assign* and *project* (as shown in Figure 2.6). We reported the issue, it was confirmed and fixed. For Octagons, all our tests passed.

$$
\begin{aligned}
poly_1 &\longleftarrow poly(-6469x_1 + x_2 + INT\_MINx_3 + INT\_MAXx_5 - 2034x_6 + 7806x_7 = 0)\\
poly_2 &\longleftarrow assign(poly_1, x_0, -6469x_3 + 9743x_5 - 7913x_6 - 1148x_7 + 8206)\\
poly_3 &\longleftarrow project(poly_2, x_4)\ //\ \top\\
poly_4 &\longleftarrow assign(poly_3, x_7, -6199x_0 + 4102x_1 - 8218x_2 + 6483x_3 + 5400x_4 + 4370x_5 +\\
&\qquad\qquad 3300x_6 - 643x_7 - 4345)
\end{aligned}
$$

FIGURE 2.6: Input causing segmentation fault for ELINA Polyhedra.

2.7 LIMITATIONS

Next, we discuss the main limitations of our work, as well as possible solutions.

Finding multiple errors. Our approach is based on grey-box fuzzers, which generate inputs that cover previously untested code. Due to the search space exploration strategy, it is thus likely that shallow bugs will be discovered first. To also identify other (possibly more complex) errors, one needs to fix the revealed issues or to remove the parts of the code responsible for them, tasks which usually require expert knowledge. These problems can be avoided by integrating our tool into the development process. Moreover, having *multiple* properties for the same domain operation (as in Figure 2.3) increases the likelihood of finding different errors.

Fuzzers. A prerequisite for our work is the existence of fuzzers for the programming languages in which the abstract domains are implemented. However, we believe this is not a severe practical limitations for many mainstream programming languages: e.g., LibFuzzer is currently integrated into Clang [50], which provides support for the C language family (used in our experiments from Section 2.5), Haskell, Lua, PHP, Python, Ruby, and Rust can be also compiled to LLVM [102], while Jazzer [171] (also based on LibFuzzer) can be used for the JVM platform. Our approach does not depend on any specific feature of the fuzzers; as our results show (Section 2.5.2), missed bugs are due to the test oracles or to the values used for the configurable parameters, and not to the fuzzers themselves.

2.8 RELATED WORK

Our approach is the first to systematically test a wide range of soundness, precision, and convergence properties on complex abstract domains. It combines several existing test case generation techniques in a novel way. On the one hand, we derive executable test oracles [16] to check high-level, mathematical properties of abstract-domain implementations. On the other hand, we incorporate ideas from boundary and random testing (step 1 of the pool construction) and from feedback-directed random testing [130] (step 2) to obtain inputs for the test oracles. A key difference with the latter is that, in our case, the fuzzer controls which elements are added to the pool, by providing the operations and their arguments.

Testing static analyzers. One way to test static analyzers is by randomly generating input programs [56]. This approach is particularly effective for testing the

robustness of the analyzers, that is, for detecting input programs that cause the analyzers to crash. To also test soundness properties, Cuoq et al. instrument the code of the analyzer under test with assertions about inferred values or relations between program variables, which are then checked against concrete executions. A similar instrumentation-based approach was proposed by Klinger et al. [97] and uses differential testing to determine if the assertions should hold or not. In contrast, our technique generates input data systematically and does not require any modifications to the implementation of the tested analyzers. Moreover, it offers stronger guarantees than [97]: our ground truth is always known, thus we do not rely on the existence of multiple analyzers to determine it. Recent work by Casso et al. [44] tests the abstract interpreters by checking if the properties they inferred statically are satisfied dynamically. However, this technique uses a random inputs generator and is thus only applicable to much simpler abstract domains (e.g., for type inference or aliasing) than those supported by our approach. The technique of Taneja et al. [165] has a similar scope with ours, i.e., testing the soundness and precision of static analyzers. However, we use mathematical properties as the test oracles, whereas [165] implemented sound and precise reference analyzers.

Analysis testing and delta debugging. Similarly to [56], Andreasen et al. [4] compare concrete executions to abstract domain elements to detect soundness and precision problems. They use delta debugging [189] to reduce the size of the input programs to report the errors concisely. In contrast, we propose a technique for automatically generating the input domain elements; our approach starts with simple elements and applies a small number of operations, generating small counterexamples by construction.

Systematically testing lattice properties. Midtgaard and Møller [120] focus on quickchecking [49] basic lattice properties of abstract interpreters. Our technique was inspired by their work; it extends the set of tested properties and, as our evaluation shows, is effective on widely-used and highly-optimized abstract domain implementations. A comprehensive experimental comparison with their tool was not possible, as the authors provide constructors and helper functions for generating ordered pairs of elements only for Intervals, but not for the complex numerical domains that we consider. Without them, many of the properties cannot be tested, as the randomly generated inputs most likely do not satisfy the preconditions. For this reason, we expect that their technique cannot significantly outperform grey-box fuzzing, which we showed to be less effective than our approach (Table 2.10). We solve the problem of generating ordered inputs by applying domain operations to the existing elements (e.g., the result of join over-approximates its operands).

Formally verified static analyzers. Interactive theorem provers such as Coq [133] have been used to verify the soundness of the *design* of static analyses (e.g., in the context of type systems [65, 144]). However, the proofs do not typically provide any guarantees about the actual *implementation*, and, thus, could still benefit from automated testing techniques like ours. The Verasco project [93] extracts executable code from verified Coq formalizations of several abstract domains. This approach

produces implementations that are correct by construction but is not yet practical for complex, highly-optimized implementations. Madsen and Lhoták [116] use symbolic evaluation (based on symbolic executions and SMT solvers) to verify the correctness, i.e., safety and soundness, of abstract-domain implementations. If the problem is undecidable, they rely on a quickchecking approach inspired by [120]. Recent work by Vishwanathan et al. [177] provides analytical proofs for the operations of the tnums (tristate numbers) abstract domain, and proposes a new multiplication algorithm which is provably sound. However, none of these works formally verifies complex numerical domains.

Unsoundness in static analyzers. Even for those analyzers that are unsound by design [23, 46–48, 114] our technique is useful to detect *unintentional* sources of unsoundness and imprecision (e.g., caused by implementation errors).

Testing program analysis tools. Abstract domains are one of the components used in modern compilers and program analyzers. Besides efforts in proving properties about such components (e.g., CompCert [110] and Verasco [93]), there is a significant body of work on using testing techniques to detect issues in compilers [103, 104, 160, 184], DSE engines [94], model checkers [191], debuggers [174], and SMT solvers [6, 31, 34, 35, 37, 117, 126, 131, 132, 141, 181, 182, 185, 186, 190].

2.9 CONCLUSIONS

In this chapter, we presented an automated testing technique for detecting soundness, precision, and convergence errors in abstract-domain implementations, which are crucial components of many static analyzers. We evaluated our approach on several complex, real-world abstract domains from two widely-used libraries for numerical analysis and demonstrated its effectiveness in finding both seeded and previously unknown errors. Moreover, we showed that our technique is applicable also to learning-based abstract interpretation transformers.

3

IDENTIFYING SOUNDNESS AND COMPLETENESS ERRORS IN SMT SOLVERS

In this chapter, we present our technique for automatically generating SMT formulas from the string theory that are satisfiable or unsatisfiable by construction, and show how these formulas with known ground truth can be used for testing state-of-the-art SMT and automata-based solvers. We also explain how our approach can be extended to support regular expressions, how it generalizes to other theories and their combinations, and how it can be used for testing MAX-SMT solvers.

3.1 INTRODUCTION

SMT solvers have a large variety of applications, from program verification and synthesis to symbolic execution and concolic testing. For all the tools that rely on them to be reliable and usable in practice, the solvers have to provide *correct results*.

For a given formula, an SMT solver returns either *sat* (together with a *model* that associates a value to each free variable and an interpretation to each uninterpreted function, such that the formula evaluates to `true`), or *unsat* (if no model exists, i.e., the formula evaluates to `false` for all possible values assigned to its free variables and for all possible interpretations for the uninterpreted functions). In the unsat case, it may also return a set of clauses that lead to a contradiction (the *unsat core*).

An SMT solver is *unsound* if it yields an incorrect result, that is, yields sat for an unsatisfiable formula or unsat for a satisfiable one. The latter case is usually called *refutation unsoundness*. The solver is also unsound if it correctly yields sat or unsat, but produces an *invalid model* (this is a case of *solution unsoundness*) or an *incorrect unsat core*, i.e., a set of clauses that are in fact satisfiable. Incorrect models and incorrect unsat cores can significantly affect the results of the applications that use SMT solvers. For example, SMT-based program verifiers construct counterexamples from the models produced by the solver, showing which contract may not hold and why. Both the models and the unsat cores are used in program synthesis, for instance, to generate programs that fulfill a given specification or to optimize the synthesis algorithm (as in [25]), respectively.

The distinction between solution and refutation (un)soundness is particularly important for the developers of SMT solvers: as automatically checking the correctness of a refutation proof is much harder than checking the correctness of a model, the developers may prioritize bug reports that expose refutation unsoundness.

Since SMT solvers support undecidable theories, they cannot always determine the satisfiability of a formula and may sometimes return *unknown*. However, a solver is *unnecessarily incomplete* if it returns *unknown* for a formula from a decidable theory, such as $F := \forall x, y\colon \text{Int} :: x = y$, which falls into Presburger arithmetic. We

```
(declare-fun t () String)
(declare-fun u () String)
(assert (= (str.replace "" t u) "a"))
(check-sat)
```

FIGURE 3.1: A sat formula that exposes an unsoundness in Z3-seq and Z3str3, written in SMT-LIB syntax [18] (with prefix notation for operators). In the rest of the chapter, we show examples in mathematical notation, to improve readability.

can also distinguish between *refutation* and *solution incompleteness* (i.e., when the solver cannot prove unsat, as for the previously-defined formula F, or it cannot find a model). It is also undesirable for a solver to *time out* (that is, not to solve the query within a given timespan). Such a result often points to performance issues.

SMT solvers use multiple interconnected decision procedures for various theories, e.g., uninterpreted functions, linear/non-linear arithmetic, bit-vectors, arrays, strings, etc. As a result, they are complex software systems and checking that their implementations are *sound* is, thus, very challenging. To illustrate the errors solvers can make, let us consider the formula from Figure 3.1. It uses two free variables of type String, t and u, and checks if it is possible to obtain the constant string $"a"$ by replacing the first occurrence of t by u in an empty string. This is the case according to the SMT-LIB standard [18] if t is empty and u is $"a"$. However, Z3-seq [28, 187] and Z3str3 [21], two widely-used SMT solvers, incorrectly report unsat.

Prior testing techniques for SMT solvers do not reliably detect such errors. Fuzzing [31] generates formulas that may crash the solvers or reveal performance issues, but do not reliably detect soundness problems. Approaches based on differential testing [119] compare the results of different solvers. Different results may indicate a soundness problem in one of them. However, determining which solver is at fault requires manual effort. Moreover, differential testing requires at least one solver that provides the correct result; this might not be the case in situations like the one described above, where Z3-seq and Z3str3 are both incorrect.

This work. In this chapter, we propose a novel technique for automatically generating test cases that reveal, besides others, *soundness* issues in the implementation of SMT solvers. We synthesize input formulas that are satisfiable or unsatisfiable by construction and use this ground truth as the test oracle. Our technique generates sequences of input formulas of increasing complexity by applying a set of satisfiability-preserving transformations that we designed. In this way, the bugs are often found with simple inputs, which facilitates error localization and debugging. We automatically construct a model for each satisfiable formula and a minimal, unique unsat core for each unsatisfiable one, and use them as witnesses or as additional oracles. For concreteness, the discussion in the following sections focuses on string solvers, but our technique generalizes to other theories and their combinations (as we show in Section 3.5.1) and can be used also for testing MAX-SMT (Section 3.5.3) and automata-based solvers (Section 3.4.3).

Contributions. The contributions of this chapter are the following:

- We present an automated approach for synthesizing SMT formulas for the string theory, which are satisfiable or unsatisfiable by construction. Together with the known ground truth, these formulas are used to automatically test the implementations of SMT solvers. Our technique generates satisfiable formulas together with models, and unsatisfiable formulas together with unsat cores; as they are incrementally complex, these formulas facilitate debugging and faster error localization.
- We implemented our technique and evaluated it on three widely-used SMT solvers, Z3-seq [28, 187], Z3str3 [21], and CVC4 [112], as well as on the automata-based solver MT-ABC [9]. Our experimental results show that our approach effectively detects soundness problems and outperforms the closest fuzzing technique for string solvers in doing so. Our work can also reveal other types of errors, such as performance, completeness, or precision issues.

Outline. The rest of this chapter is organized as follows: in Section 3.2, we give an overview of our approach for constructing satisfiable and unsatisfiable formulas from the string theory; the details follow in Section 3.3. We discuss our experimental results in Section 3.4. We show how our technique can be extended to generate formulas with regular expressions, how it generalizes to other theories, and why is applicable also to MAX-SMT solvers in Section 3.5, and explain its limitations in Section 3.6. We present related work in Section 3.7 and conclude in Section 3.8.

3.2 OVERVIEW

Our approach automatically generates SMT formulas that are satisfiable or unsatisfiable by construction. These formulas are used as inputs for black-box tests, while the ground truth represents the test oracle. Our formula construction approach consists of two steps: (1) we generate simple formulas with known truth values, and (2) from them, we derive more complex, equisatisfiable formulas through automatic transformations. To perform these steps, our generator requires a set of operations supported by the theory under test, together with their *reference semantics*. For concreteness, in this work we use the SMT-LIB standard [18] as the reference semantics because it provides a rigorous description of the theories. Most widely-used SMT solvers adhere to SMT-LIB to facilitate comparisons (e.g., in SMT competitions [151]) and to enable the side-by-side usage of multiple solvers [74].

Our technique tests if the implementation of an SMT solver complies with the provided reference semantics. For solvers that *intentionally* deviate from the SMT-LIB standard, it is straightforward to parameterize our technique with an *alternative* reference semantics and use that to test the implementation. For instance, the CVC4 documentation does not define the result of the `replace` operation when the second argument is the empty string [57]. One can use our technique with the

Return a string	Return an integer	Return a boolean
$\texttt{at}(s, off) = res^*$	$\texttt{indexOf}(s, t, off) = res$	$\texttt{contains}(s, t) = res$
$\texttt{concat}(s, t) = res$	$\texttt{length}(s) = res$	$\texttt{equals}(s, t) = res$
$\texttt{intToStr}(n) = res$	$\texttt{strToInt}\ (s) = res$	$\texttt{prefixOf}(s, t) = res$
$\texttt{replace}(s, t, u) = res$		$\texttt{suffixOf}(s, t) = res$
$\texttt{substr}(s, off, len) = res$		

*char (i.e., string of length 1) s, t, u: type String; n, off, len: type Int

TABLE 3.1: String operations, grouped by their return type.

SMT-LIB semantics to check for such deviations from the standard, and with an alternative reference semantics to check for errors in the implementation.

Table 3.1 summarizes the operations supported by the string theory, which have, according to SMT-LIB, deterministic semantics [156]. Since some of the operations take or yield integers, reasoning about them also involves linear integer arithmetic.

In the following, we give an overview of our construction approach for satisfiable (Section 3.2.1) and unsatisfiable formulas (Section 3.2.2).

3.2.1 *Generating satisfiable formulas*

An SMT formula is *satisfiable* (sat) with respect to some background theory if there exists at least one model (i.e., variable assignment) within the theory such that the formula evaluates to `true` [27]. For example, the formula $x + x = 3$ is satisfiable in the theory of real numbers, as it is possible to find at least one solution to this equation ($x = 1.5$). Nonetheless, the formula is *unsatisfiable* (i.e., does not have a model) in the integer arithmetic theory. Here and in Section 3.2.2 we use a *simplified* definition of (un)satisfiability, which does not consider the existence of an interpretation for each uninterpreted function. These are not relevant for our discussion, because we generate formulas that contain only *interpreted* functions (see Section 3.6 for more details about the applicability of our approach).

Step 1: Simple formulas. In the first step, we construct sat formulas that test each operation in isolation; this allows developers to localize and fix bugs faster. We start with the operations supported by the theory under test and automatically derive a test case for each of them, in which the parameters and the result of each operation are unconstrained. These simple formulas are thus trivially satisfiable.

An example formula that we synthesize in the first step is shown in Figure 3.2. This formula is satisfiable: for all string arguments s, t, u, there exists a string *res* that is equal to the result of the `replace` operation (all string operations are

$$\texttt{replace}(s, t, u) = res$$

FIGURE 3.2: A sat formula generated in step 1. All the variables have type String.

$$\texttt{indexOf}(s, t, \mathit{off}) = \mathit{res}$$

FIGURE 3.3: A sat formula generated in step 1, which uncovers an incompleteness in Z3-seq. *off* and *res* have type Int, *s* and *t* have type String.

total functions), thus the formula has at least one model. Even though they are very simple, these formulas can still reveal bugs. For example, Z3-seq returns unknown for the SMT formula from Figure 3.3, which tests the `indexOf` operation. Since this initial test case is minimal (and, in particular, does not involve any other operations), it facilitates identifying the source of this incompleteness: most likely a bug in handling the `indexOf` operation in the corresponding decision procedure.

Step 2: More complex formulas. To test more complex cases, as well as the interaction between different operations, step 2 of our approach derives additional test cases by automatically applying a set of transformations on the formulas synthesized before. These transformations preserve the satisfiability of the initial formulas, thus creating *equisatisfiable*, but more complex formulas, with more constrained models and more (or more complex) terms. We illustrate a very simple transformation here and present more complex ones in Section 3.3.1.

For the simple formulas from Figure 3.2 and Figure 3.3, the solver can construct arbitrarily many models, as all the variables are unconstrained. We can strengthen the formulas by adding constraints on the values of these variables. A possible transformation is to replace some of the variables with constants. To decide what values can be assigned to which variables such that the complex formula is still satisfiable, we rely on *concrete execution*. That is, we implement an *executable* version of the reference semantics for the operations under test and use it to determine valid parameters and results. This technique can be applied to sat formulas, since finding *a* model for which the formula holds is enough for proving its satisfiability.

Having the executable semantics, we can evaluate each operation on concrete arguments. In this way, we obtain formulas with constant arguments and results. The test formulas are then synthesized by fixing some of the arguments/results to the constants used in the concrete execution and leaving the others unconstrained.

Let us consider the `replace` operation from Figure 3.2. If we evaluate it on the arguments $s = ""$, $t = ""$, $u = "a"$, the result is, according to the SMT-LIB semantics, the constant string $"a"$. We can thus transform the formula by replacing the variables *s* and *res* with the constants $""$ and $"a"$, respectively. This substitution yields the sat formula from Figure 3.1, which has more constrained models and exposes a soundness bug in two widely-used SMT solvers. This and other satisfiability-preserving transformations, including some that combine multiple operations, are described in more detail in Section 3.3.1.

3.2.2 *Generating unsatisfiable formulas*

To show that a formula is *unsatisfiable* (unsat), a solver has to prove that there does not exist an assignment to the free variables within the background theory that satisfies the formula; i.e., the formula evaluates to `false` for all possible, type-correct values assigned to its variables [27].

Since many SMT theories include infinite sorts such as integers or strings, it is not possible to enumerate and check all possible value assignments and, thus, we cannot use our executable semantics to determine the ground truth. To synthesize formulas that are unsat by construction, we start from the following observation: conjoining a formula and the negation of an equivalent formula always results in an unsatisfiable formula. That is, for two equivalent formulas A and B, the formula $F := \neg A \wedge B$ is unsatisfiable by construction. We obtain interesting formulas F by leveraging equivalences between different operations from the string theory. Out of the 12 operations from Table 3.1, only `concat`, `length`, and `equals` are considered primitive operations; all the others can be expressed through them [194]. Table 3.2 shows *equivalent formulas* for non-primitive string operations and extends the preprocessing rules from [194] and the string function definitions from [8].

Note that an alternative idea would be to use a weaker condition: if $B \Rightarrow A$ holds (instead of $A \Leftrightarrow B$), then $F := \neg A \wedge B$ is also unsatisfiable. However, determining the truth value of the implication *without* using an SMT solver is not trivial. Since equivalences can be easily obtained from the literature, we use them in our work.

Step 1: Simple formulas. In the first step of our input construction technique, we automatically generate a test case from each of the 12 equivalences E1–E12 by conjoining the negation of the formula from Table 3.2, column 2 (i.e., the negation of A) and its equivalent formula from column 3 (that is, B).

The equivalences from Table 3.2 are implicitly universally quantified over their free variables. For example, E1 $:= \forall s\colon \text{String}, \mathit{off}\colon \text{Int}, \mathit{res}\colon \text{String} :: A \Leftrightarrow B$, with $A := \texttt{at}(s, \mathit{off}) = \mathit{res}$ and $B := \mathit{res} = \texttt{substr}(s, \mathit{off}, 1)$. Thus, the formulas G and H,

$$G := \forall s\colon \text{String}, \mathit{off}\colon \text{Int}, \mathit{res}\colon \text{String} :: \neg A \wedge B$$
$$H := \exists s\colon \text{String}, \mathit{off}\colon \text{Int}, \mathit{res}\colon \text{String} :: \neg A \wedge B$$

are both unsatisfiable by construction (G is stronger than H). As universal quantification poses additional solving challenges and testing various quantifiers instantiations algorithms is beyond the scope of our work, the simple unsat formulas we construct are similar to H (i.e., use existential instead of universal quantifiers).

However, we also omit these existential quantifiers from all our formulas (that is, the existentially-quantified variable becomes a fresh free variable). This is possible as all existential quantifiers are in positive positions, and $Q(x)$ is (un)satisfiable if and only if $\exists x :: Q(x)$ is (un)satisfiable. Since the resulting formulas do not use any existentially-quantified variables, we use the terms *quantifier* and *quantified variable* in the rest of the chapter to refer to *universal quantifier* and *universally-quantified variable*, respectively. For the quantifiers from Table 3.2, we specified *patterns* (also

Id	String operation	⇔	Equivalent formula
E1	$\texttt{at}(s, off) = res$	⇔	$res = \texttt{substr}(s, off, 1)$
E2	$\texttt{intToStr}(n) = res$	⇔	$(res = \text{""}$ **if** $n < 0) \wedge$
			$(res = \text{"0"}$ **if** $n = 0) \wedge \ldots \wedge (res = \text{"9"}$ **if** $n = 9) \wedge$
			$(res = \texttt{intToStr}(n \texttt{ div } 10) \texttt{ ++ } \texttt{intToStr}(n \texttt{ mod } 10)$ **if** $n \geq 10)$
E3	$\texttt{replace}(s, t, u) = res$	⇔	$i = \texttt{indexOf}(s, t, 0) \wedge (\exists s_1, s_2, s_3 :: s = s_1 \texttt{ ++ } s_2 \texttt{ ++ } s_3 \wedge$
			$\texttt{length}(s_1) = i \wedge s_2 = t \wedge res = s_1 \texttt{ ++ } u \texttt{ ++ } s_3$ **if** $i \geq 0) \wedge$
			$(res = s$ **otherwise**$)$
E4	$\texttt{substr}(s, off, len) = res$	⇔	$(\exists s_1, s_2, s_3 :: s = s_1 \texttt{ ++ } s_2 \texttt{ ++ } s_3 \wedge \texttt{length}(s_2) = len \wedge res = s_2 \wedge$
			$\texttt{length}(s_1) = off$ **if** $off \geq 0 \wedge off < \texttt{length}(s) \wedge len > 0) \wedge$
			$(res = \text{""}$ **otherwise**$)$
E5	$\texttt{indexOf}(s, t, off) = res$	⇔	$(res = off$ **if** $t = \text{""} \wedge off \geq 0 \wedge off \leq \texttt{length}(s)) \wedge$
			$(\exists s_1, s_2, s_4 :: s = s_1 \texttt{ ++ } s_2 \texttt{ ++ } t \texttt{ ++ } s_4 \wedge off = \texttt{length}(s_1) \wedge$
			$(\forall i :: \{\texttt{substr}(t, 0, i)\}\ i \geq 0 \wedge i < \texttt{length}(t) \Rightarrow$
			$\texttt{contains}(s_2 \texttt{ ++ } \texttt{substr}(t, 0, i), t) = \texttt{false}) \wedge$
			$res = \texttt{length}(s_1 \texttt{ ++ } s_2)$ **if** $t \neq \text{""} \wedge off \geq 0 \wedge off \leq \texttt{length}(s)) \wedge$
			$(res = -1$ **otherwise**$)$
E6	$\texttt{strToInt}(s) = res$	⇔	$(\texttt{intToStr}(res) = s$ **if** $s \neq \text{""} \wedge \forall j :: \{\texttt{at}(s, j)\}\ j \geq 0 \wedge$
			$j < \texttt{length}(s) \Rightarrow \texttt{at}(s, j) = \text{"0"} \vee \ldots \vee \texttt{at}(s, j) = \text{"9"}) \wedge$
			$(res = -1$ **otherwise**$)$
E7	$\texttt{contains}(s, t) = \texttt{true}$	⇔	$\exists s_1, s_3 :: s = s_1 \texttt{ ++ } t \texttt{ ++ } s_3$
E8	$\texttt{contains}(s, t) = \texttt{false}$	⇔	$\forall s_1, s_2, s_3 :: \{s_1 \texttt{ ++ } s_2 \texttt{ ++ } s_3\}\ (s = s_1 \texttt{ ++ } s_2 \texttt{ ++ } s_3) \Rightarrow (s_2 \neq t)$
E9	$\texttt{prefixOf}(s, t) = \texttt{true}$	⇔	$\exists t_2 :: t = s \texttt{ ++ } t_2$
E10	$\texttt{prefixOf}(s, t) = \texttt{false}$	⇔	$\forall t_1, t_2 :: \{t_1 \texttt{ ++ } t_2\}\ (t = t_1 \texttt{ ++ } t_2) \Rightarrow (t_1 \neq s)$
E11	$\texttt{suffixOf}(s, t) = \texttt{true}$	⇔	$\exists t_1 :: t = t_1 \texttt{ ++ } s$
E12	$\texttt{suffixOf}(s, t) = \texttt{false}$	⇔	$\forall t_1, t_2 :: \{t_1 \texttt{ ++ } t_2\}\ (t = t_1 \texttt{ ++ } t_2) \Rightarrow (t_2 \neq s)$

s, s_1, s_2, s_3, s_4, t, u: type String; n, off, len, i, j: type Int
we use mathematical equality for the `equals` operation
patterns for universal quantifiers are shown between {}
++ denotes string concatenation
`div` denotes integer division
`mod` returns the reminder of `div`

TABLE 3.2: Equivalent formulas for non-primitive string operations. The formulas E1–E12 are implicitly universally quantified over their free variables.

called *triggers*). They are used by the solver to decide when to instantiate the quantifiers via E-matching [63] and, thus, affect its ability to refute the inputs.

Our technique can be parameterized with different equivalent formulas. Other equivalences can be obtained, for example, by rewriting (sub)formulas from Table 3.2, column 3 using equalities from column 2. The formulas without quantifiers can be used also for testing SMT solvers that do not support quantification yet.

A test case that we synthesize in step 1 and corresponds to E3 is shown in Figure 3.4. For this input, CVC4 times out, and Z3str3 non-deterministically returns timeout, unsat, unknown, or segmentation fault.

$$\begin{aligned}&\neg(\texttt{replace}(s,t,u) = res) \ \wedge \ i = \texttt{indexOf}(s,t,0) \wedge \\ &(i \geq 0 \Rightarrow s = s_1 \texttt{ ++ } s_2 \texttt{ ++ } s_3 \ \wedge \ \texttt{length}(s_1) = i \wedge \\ &\qquad\qquad s_2 = t \ \wedge \ res = s_1 \texttt{ ++ } u \texttt{ ++ } s_3) \wedge \\ &(i < 0 \Rightarrow res = s)\end{aligned}$$

FIGURE 3.4: An unsat formula generated in step 1 for which CVC4 times out and Z3str3 has non-deterministic behavior. i has type Int, all the other variables have type String.

Step 2: More complex formulas. In the second step of our approach, we automatically transform the previously-generated formulas into equisatisfiable, but more complex ones. These formulas either have larger unsat cores, requiring the solver to combine more terms to derive a contradiction or contain additional terms that are not relevant for proving unsat but may complicate the solver's proof search.

To show, for example, that the formula $\forall x\colon \text{Int} :: x \neq 0 \wedge x = 0$ is unsatisfiable, the solver relies on the fact that no number can be at the same time zero and non-zero. If the conjunct $x = 0$ is replaced by $x' = 0 \wedge 2x' - x' = x$, where x' is a fresh integer variable, then all three conjuncts contribute to proving unsat. By removing any of them, the formula becomes satisfiable. Thus, these three terms represent the *minimal unsat core*. The transformations that we designed for the unsat case, described in detail in Section 3.3.2, are based on similar rewritings.

This concludes the high-level overview of our approach. Given an executable version of the reference semantics and the equivalent formulas, test case generation is deterministic and fully automatic. The ground truth is always known, so all the synthesized formulas can be directly used for testing, without additional human effort for constructing the test oracles. Our approach produces increasingly complex test cases, which often allows developers to detect errors with simple inputs, such that they are easy to reproduce and debug.

3.3 SATISFIABILITY-PRESERVING TRANSFORMATIONS

In this section, we describe our technique for constructing complex SMT formulas from the string theory through satisfiability-preserving transformations. Our transformations are different for the sat and the unsat case, therefore they are presented separately in the following subsections.

3.3.1 *Transformations for satisfiable formulas*

In the first step of the sat input construction technique (described in Section 3.2.1 and presented in pseudo-code in Algorithm 3.1, lines 2–7), we synthesize simple formulas, with unconstrained parameters and results, which are trivially satisfiable. The second step (Algorithm 3.1, lines 8–17) strengthens the initial formulas by adding constraints on the values of the free variables and synthesizes formu-

Algorithm 3.1: Our algorithm for synthesizing sat formulas. `invokeSolver` yields the solver's result on the input formula (i.e., sat, unsat, unknown, timeout, or error), a model for sat formulas, and an unsat core for unsat formulas, if available. `correctModel` uses the reference semantics to check the validity of the model with respect to the input formula. We do not check partial models (generated by some solvers for unknown results), as their correctness is not guaranteed. The auxiliary procedures `constantAssignment`, `termsPool`, and `termSynthesis` are given in Algorithm 3.2, Algorithm 3.3, and Algorithm 3.4.

Arguments : operations — tested operations (e.g., from Table 3.1)
consts — predefined constants
δ — maximum depth for term synthesis (≥ 1)

```
1  Procedure synthesizeSatFormulas
2      simpleFormulas ⟵ {}                                          // step 1
3      foreach op ∈ operations do
4          simpleFormulas ⟵ simpleFormulas ∪ {op}
5          res, model, _ ⟵ invokeSolver(op)
6          if (res ≠ SAT ∨ ¬correctModel(model, op)) then
7              reportError()
8      pool ⟵ termsPool (operations, consts, δ)                     // step 2
9      foreach F ∈ simpleFormulas do
10         foreach input ∈ constantAssignment (F, consts) do        // constant assignment
11             res, model, _ ⟵ invokeSolver(input)
12             if (res ≠ SAT ∨ ¬correctModel(model, input)) then
13                 reportError()
14         foreach input ∈ termSynthesis (F, pool, δ) do            // term synthesis
15             res, model, _ ⟵ invokeSolver(input)
16             if (res ≠ SAT ∨ ¬correctModel(model, input)) then
17                 reportError()
```

las that may contain multiple operations to test their interactions. Our algorithm is deterministic, that is, always tests the same (combination of) operations, in the same order, and on the same parameters.

We present two satisfiability-preserving transformations in the following. *Constant assignment* uses the executable semantics to compute models for simple sat formulas and then transforms them by replacing some of their free variables with values from the model. *Term synthesis* enumerates terms from the theory under test and evaluates them using the executable semantics. It then substitutes free variables from the simple formulas with more complex terms, such that the formulas remain satisfiable. As we start from quantifier-free sat formulas and none of these transformations introduces quantifiers, all our sat formulas are *quantifier-free*.

Algorithm 3.2: Auxiliary procedure for Algorithm 3.1, which generates more complex formulas by applying the constant assignment transformation on F.

```
Arguments : F — simple formula
            consts — predefined constants
Result: A set of more complex formulas
1 Procedure constantAssignment
2     complexFormulas ⟵ {}
3     foreach constArgs ∈ ⨉ typeCorrect(consts, args(F)) do
4         model[res(F)] ⟵ execute(F, constArgs)
5         model[args(F)] ⟵ constArgs
6         foreach vars ∈ ⨉ freeVars(F) do
7             input ⟵ F[model[vars]/vars]
8         complexFormulas ⟵ complexFormulas ∪ {input}
9     return complexFormulas
```

Constant assignment transformation. Many software errors are caused by the incorrect handling of corner cases. For this reason, the first transformation (shown in Algorithm 3.2) is inspired by boundary testing and consists of assigning predefined constants to (some of) the free variables of the initial formulas. The set of predefined constants (parameter `consts` in Algorithm 3.2) is configurable. We used typical boundary values in our experiments: for the variables of type String we used empty strings, strings of length one, as well as strings containing quotes, escape sequences, and non-ASCII characters in hexadecimal format. For integers, we chose a small set of valid and invalid indices and lengths, e.g., $\{-1, 0, 2\}$. The string operations cannot have other types of parameters, but some do have boolean results (Table 3.1, column 3), for which we considered both `true` and `false`.

Given the simple input formulas that we generated in step 1 (Section 3.2.1), we use concrete executions to determine their models. For this purpose, we implement an *executable semantics* for all string operations based on the *reference semantics*, i.e., the SMT-LIB specification [156]. This implementation is straightforward since most programming languages (e.g., Java) already provide string libraries that offer most of the operations. Therefore, the implementation effort mostly consists of ensuring that the semantics of these library operations and the SMT-LIB standard match. For example, the Java `String` method s.`replace`(t, u) replaces *all* occurrences of t in s by u, whereas the SMT-LIB operation replaces just the *first* occurrence. Moreover, converting any negative integer to a string in Java will yield its textual representation, whereas the SMT-LIB result will be the empty string. To handle these and other similar mismatches, we implement a *wrapper* for the Java string operations according to the SMT-LIB semantics, which represents our executable semantics.

We then exhaustively evaluate each operation on all type-correct combinations of constant arguments from the set of predefined values, which is finite and small (Algorithm 3.2, line 3). (Here and in the following algorithms, we use the notation

$\bigtimes S_i$ to represent the Cartesian product of the sets S_i.) Since all the string operations are total functions [156], the evaluation always succeeds, producing *witness* models for the simple formulas from step 1 (Algorithm 3.2, lines 4–5). The constant assignment transformation obtains new inputs by replacing some of the free variables in a simple formula (i.e., its arguments and result) by constants from the model (Algorithm 3.2, lines 6–8). As this transformation is based on valid models, it is guaranteed to yield equisatisfiable formulas.

Consider, for example, the `replace` operation, for which we already generated a simple test case during step 1 (see Figure 3.2). If the set of predefined constants includes the string $"a"$ and the empty string, then one of the concrete evaluations is $\texttt{replace}("","","a") = "a"$. We can use the computed model $s = "", t = "", u = "a", res = "a"$ to derive several new inputs. For instance, we can replace the free variables s and res by the corresponding constants from the model and obtain a new sat formula: $\texttt{replace}("", t, u) = "a"$. This formula, presented in SMT-LIB notation in Figure 3.1, revealed soundness bugs in Z3-seq and Z3str3.

We then run the solver on the transformed formula and report an error if the result is different from sat (Algorithm 3.1, lines 5–7). Otherwise, we check if the solver produced a correct model using our executable semantics (Algorithm 3.1, line 6): we evaluate the string operation on the parameters from the model generated by the solver and compare the result of the evaluation to the result from our witness model. If they are unequal, the solver is also unsound; we found such cases in our evaluation (see Section 3.4.1). Note that we cannot directly compare the two models, since our witness model is not necessarily unique.

Term synthesis transformation. To test the interactions between different operations, we transform the simple formulas from step 1 by replacing some of the free variables with more complex terms (constructed by the procedure `termsPool` from Algorithm 3.3). We use terms from the string theory, which are sufficient to supply string, integer, and boolean parameters or results for all the string operations.

Using the predefined set of constants, we synthesize all type-correct applications of string operations up to a predefined bound (parameter δ in Algorithm 3.3) and evaluate them using our executable semantics (Algorithm 3.3, lines 3–13); this produces a pool of terms (Algorithm 3.1, line 9). We then transform a simple formula from step 1 as follows (Algorithm 3.4): (1) We replace the arguments of the operation under test with terms from the pool (Algorithm 3.4, line 5). (2) We evaluate the resulting term in the executable semantics (Algorithm 3.4, line 6). (3) We replace the result variable in the simple formula with another term from the pool with the same result, which ensures that the equality holds (Algorithm 3.4, lines 7–8). (4) The complex formula used to test the solvers is then obtained by replacing the constants in the resulting equality by free variables, which yields a sat formula (Algorithm 3.4, lines 9–11). It is important to represent multiple occurrences of the same constant by the *same* free variable, to connect the operations more tightly, and to constrain the set of possible models. The transformation is applied exhaustively for all the terms from the pool, up to the bound δ (Algorithm 3.4, line 3).

Algorithm 3.3: Auxiliary procedure for Algorithm 3.1, which generates a pool of terms by applying operations from the theory under test up to the bound δ.

Arguments : operations — tested operations (e.g., from Table 3.1)
consts — predefined constants
δ — maximum depth for term synthesis (≥ 1)

Result: A pool of terms, grouped by the number of function applications used to construct them and by their actual result

```
1  Procedure termsPool
2      pool ⟵ {}
3      foreach op ∈ operations do
4          foreach constArgs ∈ ⨉ typeCorrect(consts, args(op)) do
5              term ⟵ op [constArgs/args(op)]
6              res ⟵ execute(term)
7              pool[0][res] ⟵ pool[0][res] ∪ {term}
8      foreach depth ∈ {1, ..., δ} do
9          foreach op ∈ operations do
10             foreach synArgs ∈ ⨉ typeCorrect(pool[depth − 1][args(op)]) do
11                 term ⟵ op[synArgs/args(op)]
12                 res ⟵ execute(term)
13                 pool[depth][res] ⟵ pool[depth][res] ∪ {term}
14     return pool
```

To illustrate our technique, let us assume that the set of constants includes the strings $"a"$ and $""$ and the integer -1. The pool will then contain, among others, the terms $\texttt{at}("a", -1)$ (with concrete result $""$), $\texttt{indexOf}("a","a",-1)$ (with concrete result 0), and $\texttt{concat}("","")$ (with result $""$). Starting from the simple formula for the `at` operation, the transformation proceeds as follows: (1) We substitute the arguments of `at` to obtain, e.g., $\texttt{at}(\texttt{at}("a",-1),\texttt{indexOf}("a","a",-1))$. (2) Evaluating this term yields $""$. (3) We equate the term to another term with the same result and obtain, for instance, $\texttt{at}(\texttt{at}("a",-1),\ \texttt{indexOf}("a","a",-1)) = \texttt{concat}("","")$. (4) We replace the constants $"a"$, $""$, and -1 with three fresh variables s_{fresh}, res_{fresh}, and off_{fresh}, which yields the input formula from Figure 3.5. This formula exposes a soundness bug in Z3str3, which incorrectly returns unsat. Note that when the operations `at`, `indexOf`, and `concat` were tested separately, the solver returned the expected results. It is their combination that exhibits the error.

$$\texttt{at}(\texttt{at}(s_{fresh}, off_{fresh}), \texttt{indexOf}(s_{fresh}, s_{fresh}, off_{fresh})) = res_{fresh} \texttt{ ++ } res_{fresh}$$

FIGURE 3.5: A sat formula generated in step 2 through term synthesis, which exposes an unsoundness in Z3str3. off_{fresh} has type Int, s_{fresh} and res_{fresh} have type String.

Algorithm 3.4: Auxiliary procedure for Algorithm 3.1, which generates more complex formulas by applying the term synthesis transformation on F.

```
   Arguments : F — simple formula
               pool — pool of terms
               δ — maximum depth for term synthesis (≥ 1)
   Result: A set of more complex formulas
1  Procedure termSynthesis
2      complexFormulas ⟵ {}
3      foreach depth ∈ {1, ..., δ} do
4          foreach synArgs ∈ ⨉ typeCorrect(pool[depth − 1][args(F)]) do
5              term ⟵ F[synArgs/args(F)]
6              res ⟵ execute(term)
7              foreach synRes ∈ pool[depth − 1][res] do
8                  input ⟵ term = synRes
9                  foreach c ∈ constants(input) do
10                     input ⟵ input[c_fresh/c]
11                 complexFormulas ⟵ complexFormulas ∪ {input}
12     return complexFormulas
```

The formula from Figure 3.5 was obtained with bound $\delta = 1$ for the term pool, that is, the arguments and the result are all single applications of an operation on constants; larger bounds lead to more complex formulas.

Once we have synthesized the new input, executing the test and checking if the solver returned a correct model is analogous to the constant assignment transformation (Algorithm 3.1, lines 16–18). One can easily combine the two transformations we proposed by replacing only *some* of the constants from the last step of the term synthesis transformation with free variables.

3.3.2 *Transformations for unsatisfiable formulas*

In the first step of the unsat input construction technique (described in Section 3.2.2 and presented in pseudo-code in Algorithm 3.5, lines 2–9), we test each non-primitive string operation together with its equivalent formula. Recall that if two formulas A and B are equivalent then the formula $F := \neg A \wedge B$ is by construction unsatisfiable. To obtain more complex unsat formulas, we transform the simple ones into formulas with *larger* unsat cores or with *redundant* clauses. Therefore, the solver needs to reason about more properties to prove unsatisfiability.

Consider a simple formula F that contains one variable x that occurs in both A and B, and one variable y that is existentially bound in B and, thus, becomes a free variable after the existential quantifier is removed (see Section 3.2.2). This formula can be written as:

$$F(x, y) := \neg A(x) \wedge B(x, y)$$

Algorithm 3.5: Our algorithm for synthesizing unsat formulas. `invokeSolver` yields the same results as in Algorithm 3.1. The auxiliary procedure `redundancyIntroduction` is given in Algorithm 3.6.

Arguments : operations — tested operations (e.g., from Table 3.2, column 2)
eqForms — equivalent formulas (e.g., from Table 3.2, column 3)
eqConsts — equalities between operations and constants (e.g., EC1–EC40 from Table 3.3), grouped by type
eqVars — equalities between operations and variables (e.g., EV1–EV8 from Table 3.3), grouped by type

```
1  Procedure synthesizeUnsatFormulas
2      simpleFormulas ⟵ {}                                              // step 1
3      foreach op ∈ operations do
4          input ⟵ ¬op ∧ eqForms[op]                                    // F
5          expectedCore ⟵ {op, eqForms[op]}
6          simpleFormulas ⟵ simpleFormulas ∪ {input}
7          res, _, core ⟵ invokeSolver(input)
8          if (res ≠ UNSAT ∨ core ≠ expectedCore) then
9              reportError()
10     foreach F ∈ simpleFormulas | F is ¬A ∧ B do                      // step 2
11         foreach x ∈ freeVars(A) ∩ freeVars(B) do                     // variable replacement
12             foreach eq ∈ eqVars[type(x)] | eq is SC ⇒ lhs = x do
13                 C ⟵ (lhs[x_fresh/x] = x_fresh) ∧ SC
14                 input ⟵ ¬A ∧ B[x_fresh/x] ∧ C                        // F'
15                 expectedCore ⟵ {¬A, B[x_fresh/x], C}
16                 res, _, core ⟵ invokeSolver(input)
17                 if (res ≠ UNSAT ∨ core ≠ expectedCore) then
18                     reportError()
19         foreach c ∈ constants(B) do                                  // constant replacement
20             foreach eq ∈ eqConsts[type(c)] | eq is SC ⇒ lhs = c do
21                 C ⟵ (lhs = z_fresh) ∧ SC
22                 input ⟵ ¬A ∧ B[z_fresh/c] ∧ C                        // F'
23                 expectedCore ⟵ {¬A, B[z_fresh/c], C}
24                 res, _, core ⟵ invokeSolver(input)
25                 if (res ≠ UNSAT ∨ core ≠ expectedCore) then
26                     reportError()
27         redundancyIntroduction(A, B, eqVars)                         // redundancy introduction
```

To obtain an unsat formula with a larger unsat core, we replace all the occurrences of x in B by a fresh variable x_{fresh} and conjoin a clause $C(x, x_{fresh})$ from the

string theory that implies $x = x_{fresh}$. The resulting formula is still unsatisfiable, but the unsat core now also includes $C(x, x_{fresh})$:

$$F'(x, x_{fresh}, y) := \neg A(x) \wedge B(x_{fresh}/x, y) \wedge C(x, x_{fresh})$$

Based on this general idea, we perform three transformations on the simple input formulas, which are described next and implemented in Algorithm 3.5, lines 10–27 and Algorithm 3.6. With all three transformations, the unsat core of the resulting formula is unique and known by construction. Thus, it can be used to check the correctness and minimality of the unsat core returned by the solver.

Variable replacement transformation. Our first transformation (presented in Algorithm 3.5, lines 11–18) chooses a variable x that occurs freely in A and B and constructs a more complex formula as described above. The clause $C(x, x_{fresh})$ is obtained from a set of equalities that we derived from the string theory. They are summarized in Table 3.3; we focus on the equalities EV1–EV8 here and discuss the others later. For example, the equality EV1 expresses that the first character of a string is equal to the string itself, for any string of length 1. This additional constraint about the length of the string represents a *side condition SC* under which the equality holds. We can assume that *all* the equalities from Table 3.3 have a side condition; if none is explicitly shown, it is equivalent to `true`.

EV1–EV7 are equalities on strings, while EV8 is for integers. Depending on the type of the chosen variable x, we select an appropriate equality and obtain $C(x, x_{fresh})$ by substituting the right-hand side variable by x_{fresh}, all occurrences of the same variable on the left-hand side of the equality by x, and all other variables by fresh variables (Algorithm 3.5, lines 11–13). For example, replacing the variable *res* in the formula from Figure 3.4 using EV1 yields $C(res, res_{fresh}) := \texttt{at}(res, 0) = res_{fresh}$. With this additional clause, we construct the unsat formula from Figure 3.6. Note that the side condition $SC := \texttt{length}(res) = 1$ of EV1 is *conjoined* to the formula (as shown in Algorithm 3.5, line 13), making it stronger and preserving unsatisfiability. (If we use implication instead of conjunction, i.e., $SC \Rightarrow C$ instead of $C \wedge SC$, then the resulting formula is satisfiable if $\neg A \wedge B[x_{fresh}/x] \wedge \neg SC$ holds.)

To prove that the formula from Figure 3.6 is unsat, an SMT solver can use EV1 (with *res* for s) and the conjuncts $\texttt{at}(res, 0) = res_{fresh}$ and $\texttt{length}(res) = 1$ to derive

$$\begin{aligned}
&\neg(\texttt{replace}(s, t, u) = res) \ \wedge\ i = \texttt{indexOf}(s, t, 0) \wedge \\
&\ (i \geq 0 \Rightarrow s = s_1 \texttt{ ++ } s_2 \texttt{ ++ } s_3 \ \wedge\ \texttt{length}(s_1) = i \wedge \\
&\qquad\qquad s_2 = t \ \wedge\ res_{fresh} = s_1 \texttt{ ++ } u \texttt{ ++ } s_3) \wedge \\
&\ (i < 0 \Rightarrow res_{fresh} = s) \wedge \\
&\ \texttt{at}(res, 0) = res_{fresh} \ \wedge\ \texttt{length}(res) = 1
\end{aligned}$$

FIGURE 3.6: An unsat formula generated in step 2 through variable replacement, which exposes an unsoundness in Z3str3. i has type Int, all the other variables have type String.

Id	Equality
EV1	$\texttt{at}(s, 0) = s$ **if** $\texttt{length}(s) = 1$
EV2	$\texttt{concat}(s, "") = s$
EV3	$\texttt{concat}("", s) = s$
EV4	$\texttt{replace}(s, s, s) = s$
EV5	$\texttt{replace}(s, t, u) = s$ **if** $\texttt{contains}(s, t) = \texttt{false}$
EV6	$\texttt{replace}(s, t, u) = s$ **if** $\texttt{indexOf}(s, t, 0) = -1$
EV7	$\texttt{substr}(s, 0, \texttt{length}(s)) = s$
EV8	$\texttt{indexOf}(s, "", \mathit{off}) = \mathit{off}$ **if** $\mathit{off} \geq 0 \wedge \mathit{off} \leq \texttt{length}(s)$
EC1	$\texttt{at}(s, \mathit{off}) = ""$ **if** $\mathit{off} < 0 \vee \mathit{off} \geq \texttt{length}(s)$
EC2	$\texttt{concat}("", "") = ""$
EC3	$\texttt{intToStr}(n) = ""$ **if** $n < 0$
EC4	$\texttt{replace}("", "", "") = ""$
EC5	$\texttt{substr}(s, \mathit{off}, \mathit{len}) = ""$ **if** $\mathit{off} < 0 \vee \mathit{off} \geq \texttt{length}(s) \vee \mathit{len} \leq 0$
EC6	$\texttt{intToStr}(n) = "0"$ **if** $n = 0$
...	...
EC15	$\texttt{intToStr(n)} = "9"$ **if** $n = 9$
EC16	$\texttt{indexOf}(s, t, \mathit{off}) = -1$ **if** $\mathit{off} < 0 \vee \mathit{off} > \texttt{length}(s)$
EC17	$\texttt{indexOf}(s, t, \mathit{off}) = -1$ **if** $\texttt{contains}(s, t) = \texttt{false}$
EC18	$\texttt{strToInt}(s) = -1$ **if** $s = ""$
EC19	$\texttt{strToInt}(s) = -1$ **if** $\exists i : \text{Int} :: i \geq 0 \wedge i < \texttt{length}(s) \wedge \texttt{at}(s, i) \neq "0" \wedge ... \wedge \texttt{at}(s, i) \neq "9"$
EC20	$\texttt{length}(s) = 0$ **if** $s = ""$
EC21	$\texttt{strToInt}(s) = 0$ **if** $s = "0"$
EC22	$\texttt{strToInt}(s) = 1$ **if** $s = "1"$
...	...
EC31	$\texttt{strToInt}(s) = 9$ **if** $s = "9"$
EC32	$\texttt{contains}(s, s) = \texttt{true}$
EC33	$\texttt{equals}(s, s) = \texttt{true}$
EC34	$\texttt{prefixOf}("", s) = \texttt{true}$
EC35	$\texttt{prefixOf}(s, s) = \texttt{true}$
EC36	$\texttt{suffixOf}(s, s) = \texttt{true}$
EC37	$\texttt{contains}(s, t) = \texttt{false}$ **if** $\texttt{indexOf}(s, t, 0) = -1$
EC38	$\texttt{equals}(s, t) = \texttt{false}$ **if** $\texttt{length}(s) \neq \texttt{length}(t)$
EC39	$\texttt{prefixOf}(s, t) = \texttt{false}$ **if** $\texttt{contains}(t, s) = \texttt{false}$
EC40	$\texttt{suffixOf}(s, t) = \texttt{false}$ **if** $\texttt{contains}(t, s) = \texttt{false}$

TABLE 3.3: Equalities between string operations and variables and constants of type String (EV1–EV7, EC1–EC15), Int (EV8, EC16–EC31), and Bool (EC32–EC40). All the equalities are implicitly universally quantified over their free variables.

$res = res_{fresh}$, which reduces the formula to the one we started from. This shows that the solver needs to perform additional reasoning steps, as the unsat core is *extended* by the additional conjuncts. For Figure 3.6, Z3str3 unsoundly returns sat.

This transformation is also applied to the other free variables s, t, and u, as our algorithm considers *all* free variables that appear both in A and B (Algorithm 3.5, line 11). It is also possible to replace multiple variables simultaneously, but we omitted such transformations in our experiments. There, we explore each combination of a variable and a corresponding equality (see Algorithm 3.5, lines 11–12).

Note that we can choose the granularity of the unsat core by encoding the formulas as *named assertions*. Even if some of them consist of conjuncts (e.g., C from Algorithm 3.5, line 13), the solver can include in the unsat core only an entire named assertion, not its components. Therefore, the size of the `expectedCore` from Algorithm 3.5, line 15 is 3, not 4, as we consider C to be a *single* named assertion. This design decision allows us to handle free and quantified variables uniformly (the latter are explained together with the redundancy introduction transformation).

Constant replacement transformation. Analogously to the variable replacement transformation, we can substitute a constant c by a term that evaluates to c. Starting from a simple formula $F(x, y) := \neg A(x) \wedge B(c, y)$, we construct the following formula for some constant c that occurs in B:

$$F'(x, z_{fresh}, y) := \neg A(x) \wedge B(z_{fresh}/c, y) \wedge C(c, z_{fresh})$$

To obtain the additional clause $C(c, z_{fresh})$, we use known equalities from the string theory. The equalities EC1–EC40 from Table 3.3 all equate a string term to a constant. For a chosen constant c, we select one of the type-correct equalities of the form $lhs = c$ and define $C(c, z_{fresh}) := z_{fresh} = lhs$, as shown in Algorithm 3.5, lines 19–21. As in the previous transformation, this step preserves unsatisfiability and enlarges the unsat core by the additional equality (Algorithm 3.5, line 23). This information, known by construction, is used as the test oracle (Algorithm 3.5, lines 24–26). If instead of rewriting *res*, in Figure 3.6 we replace the constant 0 by EC20 then we obtain an unsat formula that exposes a soundness bug in Z3-seq.

Note that the equalities from Table 3.3 do not contain universal quantifiers. Some of the constants could have been rewritten by using quantified formulas, but as explained in Section 3.2.2, we tried to minimize their usage as much as possible.

Redundancy introduction transformation. We also designed a variation of the variable replacement transformation, where we consider a variable y that occurs freely in B, but not in A (Algorithm 3.6, lines 2–9). These variables were initially introduced by the existential quantifiers in the equivalent formulas from Table 3.2. Consequently, renaming them to y_{fresh} and conjoining $C(y, y_{fresh})$ to the formula does not extend the unsat core; it is known by construction that y (and therefore also y_{fresh}) cannot be part of the contradiction between $\neg A$ and B, since y is not a shared variable. Nonetheless, this transformation introduces *redundancy*, that is, additional variables and terms that may obfuscate the proof of unsatisfiability.

Algorithm 3.6: Auxiliary procedure for Algorithm 3.5, which generates more complex formulas by applying the redundancy introduction transformation. `invokeSolver` yields the same results as in Algorithm 3.1.

Arguments : A — input formula (e.g., from Table 3.2, column 2)
B — formula equivalent to A (e.g., from Table 3.2, column 3)
eqVars — equalities between operations and variables (e.g., EV1–EV8 from Table 3.3), grouped by type

1 **Procedure** redundancyIntroduction
2 **foreach** $y \in$ `freeVars(`B`)` $\setminus$ `freeVars(`A`)` **do** `// free variables`
3 **foreach** $eq \in$ eqVars[`type(`y`)`] $\mid eq$ is $SC \Rightarrow lhs = y$ **do**
4 $C \longleftarrow (lhs[y_{fresh}/y] = y_{fresh}) \wedge SC$
5 input $\longleftarrow \neg A \wedge B[y_{fresh}/y] \wedge C$ `// F'`
6 expectedCore $\longleftarrow \{\neg A, B[y_{fresh}/y]\}$
7 res, _, core $\longleftarrow$ `invokeSolver(`input`)`
8 **if** (res $\neq$ UNSAT $\vee$ core $\neq$ expectedCore) **then**
9 `reportError()`
10 **foreach** $y \in$ `quantVars(`B`)` $\mid B$ is $D \wedge (\forall \overline{y} :: \{P(\overline{y})\}\ L \Rightarrow R)$ **do** `// quantified`
11 **foreach** $eq \in$ eqVars[`type(`y`)`] $\mid eq$ is $SC \Rightarrow lhs = y$ **do** `// variables`
12 $C \longleftarrow SC \Rightarrow (lhs[y_{fresh}/y] = y_{fresh})$
13 $B' \longleftarrow D \wedge (\forall \overline{y}, y_{fresh} :: \{P(\overline{y}, y_{fresh}), lhs\}$
$(C \wedge L[y_{fresh}/y]) \Rightarrow R[y_{fresh}/y])$
14 input $\longleftarrow \neg A \wedge B'$ `// F'`
15 expectedCore $\longleftarrow \{\neg A, B'\}$
16 res, _, core $\longleftarrow$ `invokeSolver(`input`)`
17 **if** (res $\neq$ UNSAT $\vee$ core $\neq$ expectedCore) **then**
18 `reportError()`

It is also possible to apply the redundancy introduction transformation to *quantified* variables from B (see Algorithm 3.6, lines 10–18); this requires changes to the quantifier's body (which we assume to have the shape $L \Rightarrow R$, as shown in Algorithm 3.6, line 10, i.e., the shape that all our quantified equivalent formulas from Table 3.2 have) and to its pattern $\{P(\overline{y})\}$. D represents the quantifier-free part of B (if absent, as in E8, E10, and E12, then D implicitly equals `true`).

We illustrate the steps of our algorithm on the formula $F := \neg(\texttt{prefixOf}(s,t) = \texttt{false}) \wedge (\forall t_1, t_2 : \text{String} :: \{t_1 \texttt{ ++ } t_2\}\ (t = t_1 \texttt{ ++ } t_2) \Rightarrow (t_1 \neq s))$ from E10, rewriting the quantified variable t_2 using EV7. The additional clause from Table 3.3 (that is, $\texttt{substr}(t_2, 0, \texttt{length}(t_2)) = t_{2fresh}$) is added under the quantifier to strengthen L, which is now expressed in terms of the fresh variable t_{2fresh}. The new variables that occur in the clause (here only t_{2fresh}) are added as universally-quantified variables and the pattern of the quantifier is extended into a multi-pattern, where the first component mentions all the quantified variables (t_1, t_2, and t_{2fresh}), while the

$$\neg(\texttt{prefixOf}(s,t) = \texttt{false}) \wedge$$
$$\forall t_1, t_2, t_{2fresh} :: \{t_1 \texttt{ ++ } t_{2fresh}, \texttt{substr}(t_2, 0, \texttt{length}(t_2))\}$$
$$((\texttt{substr}(t_2, 0, \texttt{length}(t_2)) = t_{2fresh}) \wedge (t = t_1 \texttt{ ++ } t_{2fresh})) \Rightarrow (t_1 \neq s)$$

FIGURE 3.7: An unsat formula generated in step 2 through redundancy introduction for universally-quantified variables, which exposes an unsoundness in and non-deterministic behavior for Z3str3. All the variables have type String.

second one is the `lhs` of the additional equality (i.e., $\texttt{substr}(t_2, 0, \texttt{length}(t_2))$). These modifications are shown in Algorithm 3.6, lines 12–13. The side condition SC (in our example `true`) is added as an *implication* (Algorithm 3.6, line 12). The more conservative approach that conjoins SC (used in Algorithm 3.5 for the other transformations) is not necessary, since $C := SC \Rightarrow (lhs[y_{fresh}/y] = y_{fresh})$ is added under an universal quantifier. Recall that this additional clause implies that $y = y_{fresh}$. To prove that B' is satisfiable, the solver has to consider all the combinations of values for the quantified variables. When $C = \texttt{false}$, i.e., $y \neq y_{fresh}$, the body of B' is trivially `true`, but the solver still needs to reason about the more complex case $y = y_{fresh}$, which after variable elimination leads to the original formula B we started from. As for free variables, this transformation does not increase the size of the unsat core, which remains two (see Algorithm 3.6, line 15). The resulting formula is presented in Figure 3.7 (to simplify the notation, we do not explicitly show the side condition) and exposes a soundness issue and non-deterministic behavior for Z3str3, which returns sat or segmentation fault.

Note that the three unsatisfiability-preserving transformations can be combined and all of them can be applied multiple times, to increase the complexity of the previously synthesized formulas. In our experiments, each transformation was applied *independently* and the second synthesis step was performed only once, to facilitate error localization and to avoid generating redundant tests that fail due to the same bug. Our algorithm is deterministic, it always produces the same tests.

3.4 EVALUATION

In Section 3.4.1, we present the results we obtained by applying our test case generation technique to three widely-used SMT solvers: Z3-seq (version 4.7.1), Z3str3 (version 4.7.1), and CVC4 (version 1.6). The two Z3 string solvers use different approaches: Z3-seq (the default string solver from Z3 4.7.1) encodes string operations into operations over sequences, while Z3str3 supports strings as built-in types. The experiments show that our technique is able to synthesize formulas that reveal soundness bugs in two of the three tested solvers. They also uncover completeness and performance issues. Section 3.4.2 illustrates that our approach outperforms prior fuzzing techniques for string solvers. Our work is also effective in finding bugs in other types of solvers, as we demonstrate in Section 3.4.3 on the

automata-based solver MT-ABC [9]. In Section 3.4.4 we explain how our technique differs from subsequent works on testing SMT solvers.

3.4.1 *Testing string solvers*

In the first experiment, we tested the compliance of Z3-seq's, Z3str3's, and CVC4's string operations with the semantics defined in the SMT-LIB standard [18]. In the following, we also discuss the impact of each component of our formula synthesis technique in finding the bugs.

Experimental setup. All the formulas that we synthesized are encoded, through the Z3 Java API, into SMT-LIB 2.6 format [18]; patterns are part of this standard. We used the SMT-LIB Unicode Strings Theory [156] as the reference semantics and our wrapper of the Java string operations for the executable semantics. We set a timeout of 15 seconds for each test and we fixed the seed for the solvers' random number generator, the `sat.random_seed` (for all three solvers), and the `smt.random_seed` (only for Z3-seq and Z3str3) to 0, to reduce non-determinism. We used the options `produce-models` and `produce-unsat-cores` to enable the generation of the models and of the unsat cores, respectively. We also used the option `strings-exp` for CVC4 to enable non-primitive string operations and the option `full-saturate- quant` to enable enumerative instantiation [136]. For the Z3-based solvers, we set the option `smt.core.minimize` to `true` to obtain the minimal unsat core; this option was not supported by CVC4 at the time of writing. For all other options, we used the default values; in particular, we did *not* use the solvers' own ability to check the validity of the generated models. The experiments were conducted on a 2.5 GHz Intel Core i7 CPU with 16 GB memory.

Results. The results obtained for the three solvers are summarized in Table 3.4. There, we report the expected result (column 1), the test category/transformation as described in Section 3.2 and Section 3.3 (column 2), the total number of tests generated for each of these categories (column 3), and the actual result produced by each solver (in the remaining columns). This result can be: incorrect model (when the solver correctly returned sat, but the generated model is not valid, i.e., evaluating the original formula on the model, using our executable semantics, yields `false`), incorrect unsat core (when the solver returned unsat, but the unsat core it produced is not the minimal, expected one), sat, unsat, unknown, timeout, or error (when the solver crashed or returned an error message). For the unsat formulas, we report the results for each category *without* using patterns for quantifiers and *with* the patterns specified in Table 3.2 for the formulas that are quantified (unsat^p in Table 3.4). When the patterns are not provided, the solvers infer them automatically, as E-matching [63], the algorithm used for refuting formulas, requires a pattern for every quantifier. All our sat formulas are quantifier-free.

The categories *operation* and *equivalent formula* refer to simple formulas synthesized in step 1 for testing each operation in isolation, or together with its equivalent formula from Table 3.2, respectively. The category *larger unsat core* includes the test

			#Tests with actual result (all random seeds = 0)																			
			Z3-seq							Z3str3							CVC4					
Result	**Category**	**#Tests**	**IM**	**IC**	**S**	**U**	**K**	**T**	**E**	**IM**	**IC**	**S**	**U**	**K**	**T**	**E**	**IM**	**IC**	**S**	**U**	**K**	**T**	**E**
S	operation	12	**0**	-	10	**0**	2	0	0	**0**	-	12	**0**	0	0	0	**0**	-	12	**0**	0	0	0
S	constant	4 714	**24**	-	4 158	**14**	518	0	0	**24**	-	4 580	**105**	0	5	0	**0**	-	4 714	**0**	0	0	0
S	term syn	1 394	**2**	-	842	**0**	483	67	0	**7**	-	1 027	**109**	0	133	118	**0**	-	1 394	**0**	0	0	0
U	⇔ formula	12	-	**0**	**0**	9	0	3	0	-	**0**	**0**	8	0	3	1	-	**0**	**0**	5	0	7	0
U^p	⇔ formula	12	-	**0**	**0**	9	0	3	0	-	**0**	**0**	8	0	3	1	-	**0**	**0**	5	0	7	0
U	larger ucore	268	-	**0**	**1**	153	10	104	0	-	**10**	**5**	120	12	71	50	-	**0**	**0**	89	0	177	2
U^p	larger ucore	268	-	**0**	**1**	153	9	105	0	-	**7**	**6**	120	7	78	50	-	**0**	**0**	89	0	176	3
U	redundancy	178	-	67	**0**	50	37	24	0	-	21	**10**	58	5	41	43	-	23	**0**	26	0	125	4
U^p	redundancy	178	-	67	**0**	42	36	33	0	-	22	**16**	52	8	30	50	-	25	**0**	24	0	123	6
#S failed tests (out of 6 120)					**1 110**							**501**							**0**				
#U failed tests (out of 458)					**254**							**272**							**338**				
#U^p failed tests (out of 458)					**254**							**278**							**340**				

IM = incorrect model **S** = sat **U** = unsat **U^p** = unsat formulas with patterns (from Table 3.2)
IC = incorrect unsat core **K** = unknown **T** = timeout **E** = error
constant = constant assignment redundancy = redundancy introduction ⇔ formula = equivalent formula
term syn = term synthesis larger ucore = larger unsat core **n** = #failed tests unsoundness

TABLE 3.4: Overview of our results for Z3-seq, Z3str3, and CVC4.

cases obtained by applying the variable and constant replacement transformations from Section 3.3.2. For *term synthesis*, each variable was obtained by exactly one operation (i.e., $\delta = 1$ in Algorithm 3.3 and Algorithm 3.4). For the *larger unsat core* and *redundancy introduction* transformations, each variable and constant was rewritten in one step, by independently applying all the corresponding equalities from Table 3.3. In each test case, only one of them was rewritten, with all its occurrences (as shown in Algorithm 3.5 and Algorithm 3.6).

In Table 3.4, we also include the results for a variation of the redundancy introduction transformation, which was *not* described in Section 3.3.2. It is inspired by the `let` binder from SMT-LIB [18], which is used to name sub-terms, leading to more compact formulas. This variation handles a string operation present as a subterm in B as if it was stored in a local variable and adds an additional clause from Table 3.3 that equates this sub-term (instead of a variable) to another operation.

Figure 3.8 shows a more complex unsat formula obtained by applying this transformation for the `indexOf` operation: the first line represents $\neg A$ (see E5 from Table 3.2), lines 2–9 encode its equivalent formula B (we use the boolean variables $cond_1$ and $cond_2$ to simplify the notation), and the last line is EV4 from Table 3.3, applied for the sub-term $\texttt{substr}(t, 0, i)$, whose result is a string. This additional clause trivially holds and has no influence on the unsatisfiability proof. However, it can complicate the reasoning if the solver introduces a local variable for $\texttt{substr}(t, 0, i)$, since $\texttt{substr}(t, 0, i)$ from the last line in Figure 3.8 is *not* the same as the sub-term from the body of the quantifier; the i in the last line is a free variable, while the one from line 5 is quantified. This formula is syntactically correct, as the two i variables have *different* scopes.

Soundness issues. The number of tests that failed due to soundness issues is shown in Table 3.4 with a grey background. We classify an answer as being *unsound* if the solver returned sat instead of unsat, or vice versa, or if it generated an

$$
\begin{aligned}
&\neg(\texttt{indexOf}(s,t,\mathit{off}) = \mathit{res}) \wedge \\
&\quad(\mathit{cond}_1 \Rightarrow \mathit{res} = \mathit{off}) \wedge \\
&\quad(\mathit{cond}_2 \Rightarrow (s = s_1 \texttt{ ++ } s_2 \texttt{ ++ } t \texttt{ ++ } s_4 \ \wedge\ \mathit{off} = \texttt{length}(s_1) \wedge \\
&\qquad\qquad(\forall i :: \{\texttt{substr}(t,0,i)\}\ i \geq 0 \ \wedge\ i < \texttt{length}(t) \Rightarrow \\
&\qquad\qquad\qquad\texttt{contains}(s_2 \texttt{ ++ substr}(t,0,i),t) = \texttt{false}) \wedge \\
&\qquad\qquad\mathit{res} = \texttt{length}(s_1 \texttt{ ++ } s_2))) \wedge \\
&\quad(\neg\mathit{cond}_1 \ \wedge\ \neg\mathit{cond}_2 \Rightarrow \mathit{res} = -1) \wedge \\
&\quad(\mathit{cond}_1 = (t = \text{""} \ \wedge\ \mathit{off} \geq 0 \ \wedge\ \mathit{off} \leq \texttt{length}(s)) \wedge \\
&\quad(\mathit{cond}_2 = (t \neq \text{""} \wedge\ \mathit{off} \geq 0 \ \wedge\ \mathit{off} \leq \texttt{length}(s)) \wedge \\
&\quad(\texttt{replace}(\texttt{substr}(t,0,i),\texttt{substr}(t,0,i),\texttt{substr}(t,0,i)) = \texttt{substr}(t,0,i))
\end{aligned}
$$

FIGURE 3.8: An unsat formula generated in step 2 through the variation of the redundancy introduction transformation, which exposes an unsoundness in Z3str3. *off*, *res*, and *i* have type Int, cond_1 and cond_2 have type Bool, and all the other variables have type String.

invalid model. An incorrect unsat core represents a soundness problem if the generated unsat core is *not* unsatisfiable. We observed this kind of error only for Z3str3, for the *larger unsat core* transformation and for the *equivalent formula* category. For *redundancy introduction*, the cores generated by all the solvers were always valid, but not necessarily minimal; we consider that an *imprecision*, not an unsoundness.

For CVC4, none of the test cases revealed soundness issues. By contrast, Z3str3 has the highest number of tests that fail due to soundness bugs for both sat and unsat formulas. The variation of the redundancy introduction transformation described before proved to be useful in exposing soundness errors only for Z3str3. Figure 3.8 presents an unsat formula for which Z3str3 unsoundly returns sat.

Some other example inputs for which Z3str3 produces an incorrect result have already been discussed in the previous sections. Figure 3.9 shows another type of unsoundness, i.e., a sat formula obtained through the constant assignment transformation for which the solver correctly answered sat, but generated an invalid model: $s_{\mathit{fresh}} = \text{"}3\mathit{MayMayMaZ}\text{"}$, $t_{\mathit{fresh}} = \text{"}\mathit{MayM}\text{"}$, $\mathit{off}_{\mathit{fresh}} = 1$; with these inputs, the result of `indexOf` is 1, not 0, as prescribed by the formula in Figure 3.9.

Note that Z3str3 did not support non-ASCII strings at the time of writing. Out of the 245 sat formulas unsoundly solved by Z3str3, 22 contain such strings. For a fair evaluation, we replaced them with ASCII strings and repeated the experiments. The results were the same. Moreover, the solvers use mathematical integers, while our executable semantics uses bounded integers. We manually inspected the few

$$\texttt{indexOf}(s_{\mathit{fresh}}, t_{\mathit{fresh}}, \mathit{off}_{\mathit{fresh}}) = 0$$

FIGURE 3.9: A sat formula generated in step 2 through constant assignment for which Z3str3 produces an incorrect model. $\mathit{off}_{\mathit{fresh}}$ has type Int, s_{fresh} and t_{fresh} have type String.

$$\texttt{contains}(\texttt{intToStr}(n_{fresh}), \texttt{at}(s_{fresh}, n_{fresh})) = \texttt{contains}(s_{fresh}, s_{fresh})$$

FIGURE 3.10: A sat formula generated in step 2 through term synthesis for which Z3-seq's result depends on the random seeds. n_{fresh} has type Int, s_{fresh} has type String.

models rejected by our executable semantics that contained large numbers. All of them were valid and are, thus, *not* reported as errors in Table 3.4.

Other issues. Besides soundness problems, our tests revealed various completeness, performance, and implementation errors. For instance, the unknown result points to a completeness issue. Z3-seq returned unknown for approx. 17% of our sat formulas, blaming incompleteness in the sequence theory in all 1 003 cases. We reported several failing tests and some of them have already been fixed. The timeout result suggests a performance problem, frequently observed for unsat formulas for all three solvers. Several tests also failed for Z3str3 and CVC4 due to implementation errors. For CVC4, many tests hint at completeness or performance problems for unsat formulas, both with and without patterns for quantifiers. The reason, confirmed by the developers, is that with the given patterns, many of the quantifiers are not instantiated by default through E-matching [62]. Moreover, the enumerative instantiation [136], which is used by CVC4 when E-matching saturates, does not work optimally for non-primitive string operations. We reported the problem, and for some of our test cases, it has been already fixed.

Adding patterns for quantifiers did not improve the results for Z3-seq and Z3str3. This experimental result suggests that the patterns we specified are similar to the ones automatically inferred by the two Z3-based solvers, which use the same engine for instantiating quantifiers. Another reason for the unknown result is incompleteness in the sequence theory, reported by Z3-seq for 33% of the failed tests. Z3str3 does not provide details on the reason for the incompleteness.

Our approach can be also used for discriminating between various configurations of the solvers. For example, to test their robustness, we set the random seeds to 1 465 (a value chosen arbitrarily), and we repeated the experiments. For CVC4 the results were the same. Z3-seq and Z3str3 were less robust. Figure 3.10 shows a test case for which Z3-seq correctly returned sat when the seeds were 0 but answered unknown for the seeds 1 465. Note that all the other examples from this chapter were obtained with the random seeds set to 0.

The test cases that failed in our experiments do not necessarily refer to unique bugs. This is a general problem of any testing tool and is orthogonal to our formulas synthesis technique. Several approaches have been proposed in the literature for clustering static analysis alarms [125]; we could apply these ideas to our work, to automatically group the failing tests into similarity-based clusters. Note that our synthesis algorithm reduces by construction the number of redundant test cases; step 2 applies *individual* transformations to the simple formulas generated in step 1 (see Algorithm 3.1 and Algorithm 3.5), that is, it does not chain together transfor-

	Z3-seq			Z3str3			CVC4		
#Issues	**T**	**WS**	**F**	**T**	**WS**	**F**	**T**	**WS**	**F**
closed	9	6	**5**	3	3	**3**	5	3	**1**
open	5	3	**1**	10	6	**5**	0	0	**0**

T = *total* # of issues; **WS** = *within the scope* of this work; **F** = *found* issues

TABLE 3.5: Known bugs for Z3-seq, Z3str3, and CVC4.

mations. However, we do apply step 2 even to those formulas that already lead to a failing test in step 1, as this may uncover additional bugs. For example, Z3str3 timed out during step 1 for the formulas based on E3 and E5 but was unsound for tests derived in step 2 from these inputs.

Known bugs. Due to the large number of failed tests and the complexity of the implementation of the SMT solvers, it is not feasible to manually determine how many distinct bugs we uncovered. To evaluate how effective our technique is in detecting *distinct* bugs, we assessed how many of a set of known bugs are found by our test cases. For Z3-seq and Z3str3, we considered the closed issues reported by the users from 23^{rd} May 2018 until 26^{th} January 2019, as well as all the currently open issues with the labels `string` or `z3str3` that were confirmed by the developers and do not explicitly refer to other versions than 4.7.1. Similarly, for CVC4 we considered the issues reported from 25^{th} June 2018 until 17^{th} April 2019 (both closed and still open) related to the string theory.

The known bugs are summarized in Table 3.5. From the total number of issues (column 1 for each solver), we removed the ones that are not in the scope of this work, that is, contain regular expressions, user-defined functions based on string operations, or formulas combining string operations with bit-vectors, which the approach presented so far does not support (see Section 3.5 for its extensions). We also excluded the issues explicitly caused by configurations that we do not use. We report as *found* (column 3 for each solver) only those bugs for which we could manually find a failing test case that exhibits it, based on the description from the comments or inferred from the fix. Known bugs *not* reported as found might still be detected by our test suite, but we were not able to clearly identify an appropriate test case. That is, the reported number of found bugs is a *lower* bound on the actual number. This experiment shows that our technique effectively detects bugs that occurred in actual applications of the three tested solvers, were reported and confirmed. In total, we found 15 out of 21 bugs (71%).

Sensitivity analysis. The effectiveness of our technique depends on three ingredients: (1) the set of predefined constants, (2) the combinations of operations used in a formula, needed to test their interactions, and (3) the usage of different random seeds. The manual inspection of the tests that detected the known bugs from Table 3.5 shows that all three ingredients are necessary. Finding some of the bugs required specific ways of constructing the inputs; e.g., the implementation errors from CVC4 are revealed only by the tests that use the equalities EV1 and EV5

#Failed tests	Z3-seq (4.8.6)	Z3str3 (4.8.6)	CVC4 (1.7)
S	6 [out of 1 110]	237 [out of 501]	0 [out of 0]
U	156 [out of 254]	267 [out of 272]	335 [out of 338]
$\mathbf{U}^p$	156 [out of 254]	274 [out of 278]	337 [out of 340]

S = sat; **U** = unsat; $\mathbf{U}^p$ = unsat formulas with patterns (from Table 3.2)

TABLE 3.6: Failed tests on newer versions of the SMT solvers.

from Table 3.3 to rewrite variables from E10 and E12. Similarly, a soundness bug in Z3str3 is detected only by the test that replaces the result variable from E3 using the equality EV1. Other bugs are revealed by several inputs that follow a common pattern, such as formulas obtained through constant assignment that test the `indexOf` operation with negative or out-of-bounds offset, or formulas generated through term synthesis that include `intToStr` or `strToInt` as arguments for other operations. The bug from Figure 3.10 can be observed only when testing the `contains` operation twice, with different random seeds. Our experiments do not suggest that certain equivalences (from Table 3.2) or transformations (from Section 3.3) are substantially more useful than others.

Recent improvements. As mentioned before, we reported several bugs for the three SMT solvers, many of which were confirmed or fixed. To assess the recent improvements, we re-ran the failed tests on newer versions of the solvers, i.e., 4.8.6 for Z3-seq and Z3str3 and 1.7 for CVC4. The cumulative results for all types of errors are presented in Table 3.6. In the following, we discuss our main observations, focusing on the soundness bugs.

Compared to the results from Table 3.4, summarized between square brackets in Table 3.6, the number of failing tests decreased substantially for the Z3-based solvers, at least in part due to our bug reports. For Z3-seq, no sat test still failed due to soundness bugs. For 69 unsat formulas based on `intToString`, Z3-seq returned sat; we reported 1 new soundness bug and it was confirmed. For Z3str3, 40 sat and 33 unsat tests failed due to soundness problems. Some of them correspond to open bugs, and we reported 3 additional soundness errors.

For CVC4 we did not find soundness bugs; the tests failed due to performance issues and because the minimization of the unsat cores was not yet supported. This feature was added in the meantime and the developers further improved the performance in response to our bug reports, but these changes were not yet part of the main branch at the time of writing.

All these results were obtained with respect to the SMT-LIB semantics. Even though for some operations the semantics described in the documentation of a particular solver may be slightly different, none of the bugs we reported were considered false positives by the developers; all three solvers intend to comply with the standard. Our experiments show that soundness and completeness bugs in decision procedures are not uncommon. They are due to various issues, e.g., misinterpretations of the expected semantics, flaws in the used algorithms, coding

errors, etc. These findings have implications for solver's developers, who need to systematically test for such bugs. Our work offers a technique to accomplish that.

3.4.2 *Comparison with fuzzers for string formulas*

In this subsection, we compare our technique with StringFuzz [31], a state-of-the-art test case generator for string formulas. For this experiment, we ran parts of a test suite generated by StringFuzz (from the folder generated.zip [158]) on the three SMT solvers, using the same versions as for our main experiments, i.e., 4.7.1 for Z3-seq and Z3str3, and 1.6 for CVC4. We discarded the tests for which the expected result was not specified, as for them we could not automatically classify the actual result as correct or not. In total, we included 700 tests in SMT-LIB 2.5 format, from nine categories: `lengths-short`, `lengths-long`, `lengths-concats`, `concats-small`, `concats-big`, `concats-balanced`, `different-prefix`, `regex-small`, and `regex-big`. All of them are quantifier-free and 120 tests include regular expressions (which our test cases do not contain). We used the same experimental setup as for our tool (with the random seeds set to 0) and we set the `lang` option to `smt2` for CVC4, to avoid parsing errors.

Since StringFuzz cannot check if the generated models and unsat cores are correct, we considered that a test passed when the solver answered sat or unsat, as expected. Z3-seq and Z3str3 timed out for 82 tests and returned correct results for all the others. Similarly, 74 tests failed for CVC4 due to timeout, while all the others passed. This experiment shows that StringFuzz could detect performance problems, but no soundness or completeness bugs. A reason may be that, besides regex, StringFuzz can generate only formulas with primitive string operations, which are not enough for revealing these classes of errors. In contrast, our technique also uncovered several soundness and completeness errors in the same versions of the SMT solvers, including confirmed bugs.

3.4.3 *Testing automata-based solvers*

Our technique is not specific to SMT solvers; it can also be applied for testing other types of solvers, such as automata-based solvers. In this subsection, we present the results for MT-ABC [9], an automata-based solver that performs model counting. MT-ABC supports both string and numerical constraints and classifies an input formula as satisfiable if the counted number of models is greater than 0. For some constraints, it may over-approximate the set of solutions, thus *imprecisely* answering sat for an unsat formula. Nonetheless, a *sound* implementation should not classify a sat formula as unsat.

As MT-ABC accepts as input a subset of the SMT-LIB format, we used a modified version of our tests that contains only supported features. We also replaced the escape sequence for double quotes within a string literal with the corresponding one in MT-ABC. As non-ASCII strings were not yet supported, we discarded

Result	Category	#Tests	#Tests with actual result S	U	E
S	operation	12	11	**1**	0
S	constant	3 568	3 145	**275**	148
S	term syn	1 394	974	**175**	245
U	⇔ formula	6	4*	2	0
U	larger ucore	121	63*	56	2
U	redundancy	71	36*	35	0
	#S failed tests (out of 4 974)			**844**	
	#U failed tests (out of 198)			**105**	

S = sat; **U** = unsat; **E** = error
constant = constant assignment
larger ucore = larger unsat core
redundancy = redundancy introduction
n = #failed tests unsoundness
n* = #failed tests imprecision
⇔ formula = equivalent formula
term syn = term synthesis

TABLE 3.7: Overview of our results for MT-ABC.

the sat tests that included these constants in the formulas or in the possible model. The unsat tests based on E2, E5, E6, E8, E10, and E12 from Table 3.2 could not be handled by MT-ABC because it did not support quantifiers and the `mod` operator, so we removed them as well. We tested the code version [123], using default options. This commit includes a fix for E7, E9, and E11, based on a crash that we found and reported. We used the SMT-LIB Unicode Strings Theory [156] as the reference semantics because MT-ABC recently updated the implementation of the string operations to match this standard.

The results are summarized in Table 3.7. The soundness issues are shown with a grey background, while the imprecisions are marked with *. As it can be observed, our technique effectively synthesized formulas that exposed various soundness and precision issues, as well as implementation failures for different string operations. For example, MT-ABC unsoundly returned unsat for the formulas from Figure 3.3 and Figure 3.5. We have already reported 6 distinct soundness bugs and 6 crashes for *constant assignment* and for the *operation* category, since they are easier to debug. The developers confirmed them and appreciated that we sent simple formulas that expose the bugs. As a result, some of them were fixed within a day.

3.4.4 *Subsequent work on testing SMT solvers*

Testing SMT solvers is a topic that has received a lot of attention from the research community in the last two years. Next, we discuss our work in the context of concurrent or subsequent research efforts in this area. These are presented in Table 3.8, columns 2–10, and are grouped by the testing techniques on which they are based.

For the comparison, we considered the following features (Table 3.8, column 1): (1) the ground truth of the generated formulas is known by construction; (2) for sat formulas, the approach can also generate a witness model; (3) the model provided by the solver can be validated by the technique itself, without using an SMT

	Differential testing				Metamorphic testing	Fuzzing			Semantic fusion	Our
Feature	**[190]**	**[181]**	**[131]**	**[132]**	**[186]**	**[141]**	**[117]**	**[185]**	**[182]**	**work**
known ground truth	–	–	–	–	✓	–	✓	–	✓	✓
witness model	–	–	–	–	–	–	–	–	≈	✓
model validation	–	–	–	–	–	–	–	–	–	✓
expected unsat core	–	–	–	–	–	–	–	–	–	✓
no seed formulas	–	–	–	–	–	–	–	✓	–	≈
no minimization	–	–	–	–	–	–	–	–	–	✓
all logics/theories	✓	✓	✓	✓	✓	–	✓	✓	–	≈
stable versions	–	–	–	–	–	✓	–	–	–	✓
more configurations	–	✓	–	✓	–	–	✓	✓	–	–
coverage/traces	–	✓	–	✓	✓	–	✓	✓	✓	–

✓ = fully supported ≈ = partially supported – = not supported

TABLE 3.8: Comparison between our work and recent techniques for testing SMT solvers.

solver; (4) for unsat formulas, the minimal expected unsat core is known; (5) the approach does not require seed formulas; (6) no minimization of the formulas is necessary; (7) the technique generates formulas from all the SMT logics/theories; (8) the approach was evaluated on stable (release) versions of the solvers; (9) the work considers various options/configurations of the solvers; (10) the technique analyzed the coverage/execution traces achieved by the generated test suite.

Our technique has the features (1)–(4), (6), and (8). It tests the solvers using standard configurations (9), does not consider coverage nor execution traces (10), and does not generally require seed formulas (5) (but for the unsat case it starts from a set of given equivalences, which can be seen as seeds). The examples presented so far and our evaluation from Section 3.4.1 focus on the string theory (7). However, Section 3.5 discusses how our work generalizes to other theories and Section 3.6 presents possible ways of supporting uninterpreted functions and quantifiers for sat formulas (some of our unsat formulas are already quantified). In the following, we explain in more details the comparison from Table 3.8.

Differential testing. Concurrent work by Zhang [190] (Table 3.8, column 2) and subsequent work by Winterer et al. [181] (Table 3.8, column 3) introduce type-aware mutation, a technique that generates inputs for differential testing by replacing operators of the original formulas with other operators of conforming types. This approach is further generalized in [131, 132] to generative type-aware mutation (Table 3.8, columns 4–5). However, it does not have a test oracle (i.e., it uses two SMT solvers for cross-checking the results), relies on the solver's internal checks for validating the models, and requires seed formulas. Our technique synthesizes simple formulas and then increases their complexity through satisfiability-preserving transformations. Directly comparing our work with [131, 132, 181, 190] in terms of the number of bugs found can be misleading: we report a lower bound for the number of unique bugs found in *one* stable (release) version of each tested solver, while the related works consider *all* the commits performed within a time span (e.g., from September 2019 to September 2020 in [181]). Moreover, their eval-

uations use formulas from arbitrary (combinations of) logics, whereas our experimental results refer only to the theory of strings, without regular expressions.

Metamorphic testing. It addresses the oracle problem through metamorphic relations [164], which for SMT solvers are usually satisfiability-preserving transformations; thus a bug is found if the solver returns different results for the original and the modified input (the mutant). Recent work by Yao et al. [186] (Table 3.8, column 6) uses as mutants equisatisfiable over- and under-approximations of the seed formulas, thus the ground truth is known by construction. These are computed based on predefined mutation rules, which are somewhat similar to our equivalences from Table 3.2 and Table 3.3. Moreover, we automatically synthesize seed inputs with a known expected output, while [186] starts from existing ones.

Fuzzing. Recent work by Scott et al. [141] (Table 3.8, column 7) combines fuzzing with reinforcement learning to generate, based on a given input grammar, quantifier-free SMT formulas from the string and floating-point arithmetic theories that can expose performance issues. Our approach mainly targets soundness problems, but can reveal also performance, completeness, and implementation errors. Subsequent work by Mansur et al. [117] (Table 3.8, column 8) proposes a black-box mutational fuzzing technique, which generates satisfiable formulas from seed inputs. As opposed to our approach, it considers only one of the four types of unsoundness, i.e., the case where the solver returns unsat for a satisfiable formula. We also construct unsatisfiable formulas and check the validity of the models and of the unsat cores produced by the solvers. Recent work by Yao et al. [185] (Table 3.8, column 9) generates seed formulas from a context-free grammar and considers the fuzzing space two-dimensional: besides the formulas, it mutates also those solver's configurations that might be relevant for a particular input. Our technique focuses on default configurations that allow models and unsat cores construction and minimization, but our formulas with known ground truth can be used also with different options (e.g., with a new random seed, as in Section 3.4.1).

Semantic fusion. Subsequent work by Winterer et al. [182] (Table 3.8, column 10) introduces semantic fusion, a technique that combines two equisatisfiable input formulas into a new, equisatisfiable one. For this, it relies on manually-provided fusion and inversion functions, in the same way in which we start from equivalent formulas when generating unsat inputs. The fusion functions can also construct a model for each sat formula, from the models of the two fused components (assuming these are given). Similar to [117, 131, 132, 181, 190], the approach requires seed formulas, and additional minimization techniques once a bug has been found. Our work produces increasingly complex formulas; this allows the developers to understand and fix the errors faster.

Overall, we believe that our technique offers stronger guarantees than the subsequent approaches, as it synthesizes formulas that are satisfiable or unsatisfiable by construction, together with models or with minimal unsat cores. It provides also an independent mechanism (i.e, which does not rely on an SMT solver) for

model validation and does not require minimization. However, all the works have been effective in detecting previously-unknown bugs, due to their complementary strengths. The developers of SMT solvers might thus benefit even more from a combination of techniques. In particular, our work could be used to generate seed formulas for semantic fusion. Moreover, our idea of leveraging the concrete execution for obtaining a witness model could be used to construct the test oracle (for the sat case) by the fuzzing and type-aware mutations techniques that require a second solver for checking the returned results.

3.4.5 *Threats to validity*

We identified four threats to the validity of our experiments:

Ground truth. Our technique relies on an executable semantics for the string operations, on the equivalent formulas from Table 3.2, and on the equalities from Table 3.3. Errors in these components could lead to incorrect tests. To mitigate this threat, we carefully reviewed all the components of our approach. Since they are simple variations of the SMT-LIB semantics, we are confident that they are correct.

Non-deterministic behavior. The solvers use randomized algorithms, which can lead to non-deterministic behavior (see Section 3.2.2 for an example). We mitigated this problem by fixing the random seeds and by performing the experiments from Section 3.4.1 with two different values. Nevertheless, the Z3 Java API that we use for generating the SMT files can non-deterministically assign names to temporary variables (from `let` expressions). Since for some solvers these names can influence the internal heuristics used for solving, some of the results from Table 3.4, Table 3.6, and Section 3.4.2 may require multiple runs to be reproduced.

Pattern selection. The patterns chosen for the quantified formulas are independent of the way in which each solver handles quantifier instantiations. Other patterns could have been more efficient for proving that certain formulas are unsat or alternative rewritings of the formulas could have better triggered particular instantiations. Nonetheless, our patterns are configurable and we ran the experiments with and without patterns to assess their impact.

Known bugs. To determine if our technique generates test cases that can detect known bugs, we manually matched some of the failed tests against the confirmed bugs. As we reported only clear matches as found bugs, we are confident that Table 3.5 shows a *lower* bound on the number of known bugs we detected.

3.5 EXTENSIONS

Next, we explain how our technique can be extended to generate formulas from other theories (e.g., fixed-size bit-vectors, arrays, etc.) and from combinations of theories (Section 3.5.1), as well as formulas with regular expressions (Section 3.5.2). We also show how our work can be used to test MAX-SMT solvers (Section 3.5.3).

3.5.1 *Other theories*

The main ingredients of our technique are not specific to strings. One can construct quantifier-free sat formulas from any theory that has an executable semantics (e.g., integers). If the operations are total functions and all their values are specified by the standard, then the model validation can be also done automatically (partial functions and operations such as real division ("/") pose additional challenges in checking the correctness of the model; e.g., division by 0 is defined, by its result is not specified [18]). Quantified logics and logics that allow uninterpreted functions are supported by our sat approach in a limited way, as we explain in Section 3.6.

Moreover, unsat formulas can be generated as long as operations or constants from the theory under test can be expressed in multiple forms. For instance, the fact that x is a positive real number can be expressed as $x \geq 0$ but also as $\exists y\colon \text{Real} :: x = y * y$ (every positive real number has a square root). The constant 0 can be also defined through operations from the theory of reals: $\forall z\colon \text{Real} :: 0 = z + (-z)$. From these equalities, one can generate the formula $F := \neg(x \geq (z + (-z))) \wedge (x = y * y)$, which is unsatisfiable by construction (we omitted the quantifiers to simplify the notation). F has the shape $\neg A \wedge B$ from Section 3.2.2, with $A \Leftrightarrow B$. Similar rewritings or equalities do exist for other theories.

Bit-vectors and arrays. Together with Becker, we showed in his Bachelor thesis [20] that our technique can be extended to fixed-size bit-vectors and arrays and can also generate formulas that combine multiple theories. Supporting bit-vectors is mostly straightforward, both for the sat and the unsat case: the Java `BitSet` class contains most of the SMT-LIB bit-vector operations [154] (with minor differences in semantics that we accounted for) and one can write equivalent formulas and equalities similar with those from Table 3.2 and Table 3.3 (see Table 2 and Table 4 from [20]). However, the SMT-LIB arrays [152] cannot be mapped to Java arrays, since they are more generic and allow keys of arbitrary types (not only natural numbers, as in Java); we can thus use `Maps` to represent them in Java and to evaluate the sat formulas on concrete values [20]. Nevertheless, the fact that SMT-LIB defines only two operations for arrays, `select` and `store` [152], poses additional challenges in generating unsatisfiable formulas.

We addressed them in [20] by adapting the sat constant assignment transformation (from Section 3.3.1) for synthesizing unsat formulas: we first evaluate each simple formula F generated in step 1 on a set of predefined constants (as shown in Algorithm 3.2, lines 4–6), then we construct more complex formulas of the form $F[\texttt{constArgs}/\texttt{args}(F)] = c$, where c is a constant from the predefined set *different*

$$\texttt{store}([\texttt{true} \to \texttt{true}, \texttt{false} \to \texttt{true}], \texttt{false}, \texttt{false}) = [\texttt{true} \to \texttt{false}, \texttt{false} \to \texttt{false}]$$

FIGURE 3.11: An unsat formula generated in step 2 through the unsat constant assignment transformation (defined in [20]), for which Z3 unsoundly returns sat.

$$\texttt{store}([0000 \to \text{"}a\text{"}, 0001 \to \text{"}a\text{"}, \ldots, 1111 \to \text{"}a\text{"}], 0000, \text{""}) =$$
$$\texttt{store}([0000 \to \text{""}, 0001 \to \text{""}, \ldots, 1111 \to \text{""}], 0010, \text{"}a\text{"})$$

FIGURE 3.12: An unsat formula generated in step 2 through the hybrid unsat constant assignment – term synthesis transformation (defined in [20]), which uses fixed-size bit-vectors as keys and Strings as values in the array. For this formula, Z3 unsoundly returns sat.

from the value obtained through the concrete execution. This modified transformation yields formulas *without* any free variables (i.e., in which all the parameters and the result are fixed). Figure 3.11 shows an example input generated by this transformation. The right-hand side represents a constant array with Boolean keys, where all the values are `false`. The left-hand side is obtained by storing the value `false` at the key `false` in an array where all the values were initially set to `true`. Since the equality does not hold, the formula is unsatisfiable. However, Z3 (version 4.8.12) unsoundly returns sat (see Chapter 5 from [20] for additional results).

Similarly, we also defined a hybrid unsat transformation that combines constant assignment with term synthesis. This transformation generates unsatisfiable formulas with more complex results, by equating $F[\texttt{constArgs}/\texttt{args}(F)]$ with terms from the theory under test (that is, sequences of operations applied to constants, not only constants, as before), whose concrete values are *different* from the result of evaluating F on `constArgs` [20].

Combinations of theories. Strings, bit-vectors, arrays, booleans, and integers can be directly combined, as all these theories define operations [152–156] that can generate arguments or results for the operations available in the other theories, e.g., the strings `length` operation produces an integer (see Table 3.1), which can be used as the first or second argument to the bit-vectors `extract` operation. Table 8 from [20] provides an overview of the operations, grouped by their return type. They can be used for the sat case (during term synthesis), as part of the constant/variable rewritings for unsat formulas, or during the hybrid transformation [20]. Moreover, since the arrays map arbitrary key types to arbitrary value types, the `select` operation can return *any* specific type.

Testing the SMT solvers with formulas containing terms from multiple theories can reveal bugs that cannot be detected by considering each theory in isolation. Such a formula that uncovered an unsoundness in Z3 (version 4.8.12) is given in Figure 3.12. It maps bit-vectors of length 4 to strings and was obtained by applying the hybrid transformation described above [20]. The bug was confirmed and fixed.

Figure 3.13 shows an example that combines arrays, bit-vectors, booleans, and strings, for which CVC4 (version 1.8, i.e., the latest version) generated an invalid model: $a_{fresh} = [0000 \to \texttt{false}, 0001 \to \texttt{true}, 0010 \to \texttt{false}, \ldots, 1111 \to \texttt{false}]$ (all the positions, except from 0001, are set to `false`), $b_{fresh} = 0000$, $e_{fresh} = \texttt{false}$, $s_{fresh} = \text{""}$; with these value, the left-hand side evaluates to `false`, while the right-hand side is always `true`. The error has been fixed in CVC5 [59] (the successor of CVC4 1.8), without an explicit bug report [20].

$$\texttt{select}(\texttt{store}(a_{fresh}, b_{fresh}, e_{fresh}), \texttt{bvnot}(b_{fresh})) = \texttt{contains}(s_{fresh}, s_{fresh})$$

FIGURE 3.13: An unsat formula generated in step 2 through term synthesis, which uses fixed-size bit-vectors as keys and booleans as values in the array. a_{fresh} has type Array[BitVector4, Bool], b_{fresh} has type BitVector4, e_{fresh} has type Bool, and s_{fresh} has type String. For it, CVC4 produces an incorrect model.

3.5.2 *Regular expressions*

The technique presented so far focuses on testing the implementation of the string operations (from Table 3.1) but does not consider *regular expressions (regex)*, which are also defined within the theory of strings [156]. Regular expressions are particularly important for security applications (e.g., they are used by cloud policy languages, such as [11]), thus it is crucial that the SMT solvers handle them soundly.

However, this is not always the case in practice. Figure 3.14 shows a satisfiable formula for which Z3-seq (version 4.8.14) and Z3str3RE [22] (a new solver for regular expressions) return unsat. The `range` operation produces the set of singleton strings whose ASCII values are within the ASCII values of the given bounds [156]. Since ASCII("*b*") > ASCII("*a*"), the result is the empty set of strings (represented through the constant regex `re.none`). Our transformations can be extended to generate formulas with regular expressions, as well as formulas that combine regex, strings, arrays, and bit-vectors. For instance, the input from Figure 3.14 was obtained by applying the constant assignment transformation to the `range` operation.

We explored these ideas together with Kühne, in his Bachelor thesis [99]; there we proposed two alternative approaches for constructing a pool of representative regular expressions (which are used by the sat transformations) and defined a set of equivalent formulas for the unsat case. Moreover, we used the `dk.brics.automaton` package [170] for implementing the executable semantics, as two regular expressions are equal if they describe the same regular language, but this comparison is imprecise on their string representation from standard Java. The package performs the equality check on the level of their corresponding finite-state automata. Our experiments show (see Chapter 6 from [99]) that state-of-the-art SMT solvers do not have good support yet for variables of type regular expression and for regex equality (e.g., CVC5), or they often produce incorrect models (e.g., Z3str3).

Note that it is also possible to design a *generalized* hybrid unsat transformation, which combines constant assignment and term synthesis in a more generic way than our transformation from [20]: the arguments and the result of an operation

$$\texttt{range}(\text{"}b\text{"}, \text{"}a\text{"}) = \texttt{re.none}$$

FIGURE 3.14: A sat formula generated in step 2 through constant assignment, which exposes an unsoundness in Z3-seq and Z3str3RE. `re.none` is a constant regular expression denoting the empty set of strings (i.e., the empty language).

$$\texttt{string\_in\_regex}(\texttt{""}, \texttt{repeat}(189, 0, \texttt{string\_to\_regex}(\texttt{""}))) = \texttt{true}$$

FIGURE 3.15: An unsat formula for which Z3 unsoundly returns sat. It can be obtained in step 2 through the generalized hybrid unsat transformation.

are either constants or sequences of operations over constants; to ensure unsatisfiability, the result is chosen such that its concrete value is always *different* than the concrete value of the evaluated operation. This transformation allows us to synthesize formulas similar to the one from Figure 3.15, which checks if the empty string is included in the regular language described by the regex which repeats the empty string at least 189 and maximum 0 times. According to SMT-LIB [156], the repetition where the lower bound is higher than the upper bound yields the empty language. The formula is thus unsat, but Z3 (version 4.8.7) unsoundly reports sat. However, we did not include this transformation in our evaluation from [99].

3.5.3 *MAX-SMT solvers*

Our technique is also applicable for testing a different class of SMT solvers, namely MAX-SMT solvers, which are designed for optimization problems [29]. These solvers distinguish between *hard* constraints (which have to hold) and weighted *soft* constraints (which may hold). Given an input formula consisting of both hard and soft constraints, the task of a MAX-SMT solver is to: (1) determine if the hard constraints are satisfiable, (2) if they are, then to also identify a subset of the soft constraints which hold together with the hard constraints, such that the sum of the corresponding weights is maximized. As opposed to a regular SMT solver, a MAX-SMT solver produces an invalid model (i.e., is unsound) also when there exists a different subset of the soft constraints than the one identified by the solver, whose some of the weights is the maximal sum of the satisfiable soft constraints.

Automatically testing MAX-SMT solvers thus requires inputs that contain hard and weighted soft constraints; for sat formulas, the maximum sum of the weights has to be known by construction, as this represents an additional test oracle.

All the formulas presented so far in this chapter contain only hard constraints. In the following, we briefly describe how our transformations from Section 3.3.1 and Section 3.3.2 can be used to also generate formulas with soft constraints. Note that the absence of a hard constraint is equivalent to a `true` hard constraint. For this reason, formulas that have only soft constraints are by construction satisfiable.

Sat formulas with soft constraints. A possible way to synthesize such inputs is to encode our simple formulas from step 1 as hard constraints and the sat transformations as soft constraints with *positive* weights (MAX-SMT solvers support also negative weights, which represent the penalty they have to pay for finding models that satisfy the corresponding soft constraints). Following this approach, we obtain MAX-SMT formulas where *all* the constraints (both hard and soft) are satisfiable together. Figure 3.16 illustrates the MAX-SMT version of our motivating example

$$(\texttt{replace}(s,t,u) = res)_{(H)} \wedge (s = \text{""})_{(S,w_1)} \wedge (res = \text{"}a\text{"})_{(S,w_2)}$$

FIGURE 3.16: A sat MAX-SMT formula generated by encoding the simple formula from Figure 3.2 as hard constraint (H) and the constant assignment transformation as soft constraints (S). $w_1, w_2 \in \mathrm{N}^*$ are the weights of the soft constraints. All the variables have type String.

from Figure 3.1. (For presentation purposes, we mark the hard constraint explicitly but in SMT-LIB [18] any constraint not declared as soft is considered hard.) This formula is satisfiable, with a possible model $s = \text{""}$, $t = \text{""}$, $u = \text{"}a\text{"}$, $res = \text{"}a\text{"}$ (obtained as before, through concrete execution); the expected maximum sum of the satisfiable soft constraints is $w_1 + w_2$, for any actual weights $w_1, w_2 \in \mathrm{N}^*$.

Unsat formulas with soft constraints. A MAX-SMT formula is unsatisfiable only if the hard constraints are unsatisfiable. Thus to preserve the known ground truth, the complex formulas obtained in step 2 through the constant/variable replacement transformations and the redundancy introduction transformation for quantified variables have to be encoded as hard constraints. The additional clause C from the redundancy introduction transformation for free variables (see Algorithm 3.6, line 4) can be represented as both hard and soft constraints (the choice determines the size of the unsat core). To increase the complexity of the resulting formulas, we can include additional satisfiable soft constraints, generated as explained for the sat MAX-SMT case. Figure 3.17 shows an example, where the *res* variable from E1 was rewritten using EV1. The soft constraints do not influence the satisfiability of the input, as the hard constraints are by construction unsatisfiable.

Starting from this technique, we defined together with Thommen in his Bachelor thesis [172] new transformations, which produce more complex formulas (e.g., sat formulas consisting only of soft constraints or containing both satisfiable and unsatisfiable soft constraints, to check the solver's ability to identify the optimal subset, etc.). We are currently also exploring ways to extend our approach to synthesize formulas that model *multi-objective* problems (the ones from Figure 3.16 and Figure 3.17 are *single objective*) [172].

$$\begin{aligned}&(\neg(\texttt{at}(s, \mathit{off}) = res))_{(H)} \wedge (res_{\mathit{fresh}} = \texttt{substr}(s, \mathit{off}, 1))_{(H)} \wedge \\ &(\texttt{at}(res, 0) = res_{\mathit{fresh}} \wedge \texttt{length}(res) = 1)_{(H)} \wedge \\ &(s = \text{""})_{(S,w_1)} \wedge (res = \text{"}a\text{"})_{(S,w_2)}\end{aligned}$$

FIGURE 3.17: An unsat MAX-SMT formula generated by encoding a complex formula obtained through variable replacement as hard constraints (H). The soft constraints (S) increase its complexity. $w_1, w_2 \in \mathrm{N}$ are the weights of the soft constraints. *off* has type Int, all the other variables have type String.

3.6 LIMITATIONS

Next, we explain the limitations of our technique and discuss possible solutions.

Quantified sat formulas. Our sat transformations are based on concrete execution and ensure that each complex formula evaluates to `true` for *at least one* set of concrete values. However, this allows us to synthesize only quantifier-free formulas; universally-quantified formulas would require stronger guarantees, which cannot be provided for variables of infinite types (such as String). For instance, the truth value of the formula $\forall s$: String :: $\texttt{length}(s) \geq 0$ cannot be determined automatically, as we cannot evaluate it for all possible strings s. Nonetheless, our technique is applicable to quantification over *finite* types (e.g., Boolean, bit-vector of a predefined size, etc.), since for these types one can efficiently enumerate all the combinations of values for the free variables.

Uninterpreted functions. Our synthesized formulas contain only *interpreted* functions from the theories under test. However, SMT-LIB supports also *uninterpreted* (user-defined) functions and types. Operations over variables of uninterpreted types are usually expressed through uninterpreted functions; as they do not have an executable semantics, our technique for the sat case is not directly applicable. However, a formula is satisfiable if, besides concrete values for the free variables, there exists *an* interpretation for each uninterpreted function such that the formula evaluates to `true`. We could thus *fix* the interpretation of an uninterpreted function to the interpretation of a particular interpreted function and generate formulas that assert properties about it. For instance, we could generate the sat formula $\texttt{f}(\text{"}a\text{"}) = 1$, where $\texttt{f}$: String $\rightarrow$ Int is uninterpreted. In our witness model, `f` can have the same interpretation as `length`.

For the unsat case, we could use the mathematical definition of a function and generate simple unsatisfiable formulas of the form $F := (x = y) \wedge \neg(\texttt{f}(x) = \texttt{f}(y))$. The solver cannot find an interpretation for `f`, since there does not exist a (deterministic) function which applied to the same arguments, returns different results. If x and y are variables of built-in types, we could then derive more complex formulas by applying the variable replacement transformation from Section 3.3.2. We could also define new transformations, which introduce additional uninterpreted functions, and produce, e.g., $F := (\texttt{f}(x) = \texttt{f}(y)) \wedge \neg(\texttt{g}(\texttt{f}(x)) = \texttt{g}(\texttt{f}(y)))$.

3.7 RELATED WORK

The developers of SMT solvers usually create their own test suites, which include manually-written tests and regression tests derived from bug reports [58, 188]. Our approach automates parts of this time-consuming process by generating test cases of incremental complexity; this facilitates debugging and faster error localization. Moreover, our experiments demonstrate that our technique is applicable also to automata-based solvers, which was not shown by any other related work.

In the following, we present the research efforts not considered in Section 3.4.4.

Differential testing. A common approach used in practice is differential testing [119], which compares the results of different solvers (or of different versions of the same solver) on a set of benchmarks [21, 112]. Different results suggest a bug in one of the solvers. However, determining which one is at fault requires additional effort. In our case, the ground truth is known upfront, so our synthesized input formulas can be directly used for testing, without requiring a reference implementation as a test oracle. As opposed to differential testing, our technique can be also applied when there only exists one implementation of a given semantics.

Fuzzing. Other test case generation techniques are based on fuzzing. Brummayer et al. apply fuzzing for testing SMT [34], SAT, and QBF solvers [35], while Cyrille et al. [6] and Niemetz et al. [126] target the solver's API. Since [34] generates only quantifier-free formulas over fixed-size bit-vectors, a direct comparison with our work for string solvers would not be meaningful. However, all these approaches generate inputs that may cause the solver to crash or may exhibit performance issues. As opposed to our proposed solution, they do not have a test oracle, so they do not reliably detect soundness and completeness bugs. The existing techniques require delta debugging [189] to minimize the inputs that lead to a failure; in our case, the errors are usually found with formulas that are small by construction.

The closest related work to ours is StringFuzz [31], a state-of-the-art fuzzer and generator of SMT-LIB instances. StringFuzz can create input formulas with various properties (e.g., predefined number of variables, configurable depth of expressions, predefined length for string literals, etc.), but it has generators only for primitive string operations and for regular expressions. Our evaluation does not consider regular expressions. However, it focuses also on formulas that cover complex string operations. StringFuzz can also apply a set of transformations on already existing benchmarks, but very few of them guarantee equisatisfiability. Using these formulas for soundness testing requires manually-written test oracles. In contrast, our synthesized formulas are sat or unsat by construction, and all our transformations preserve their satisfiability, therefore soundness testing is fully automatic.

Formal verification. Formal verification has been used to verify SAT and SMT algorithms [75, 111, 118], but not their implementations. SMT solvers are complex, highly-optimized software systems, thus formally verifying their implementation is very challenging. In contrast, our black-box testing technique can handle such complex implementations and can find bugs with minimal effort.

Validation and proof checking. A complementary body of work focuses on checking the proofs generated by the solvers. Zhang and Malik [192] synthesize a checker for validating the traces produced by a SAT solver during refutation proofs. Böhme and Weber [32] encode Z3's proofs in Isabelle, while Stump et al. [159] propose a meta-logic for describing and checking proofs for SMT. All these techniques require either modifications of the original solvers or translations of the proofs into other formats. Our approach treats the solvers under test as black boxes and does not depend on a specific implementation or proof format.

3.8 CONCLUSIONS

In this chapter, we presented a novel technique for automatically generating SMT formulas from the string theory that are satisfiable or unsatisfiable by construction. They are used as inputs for testing mostly the soundness of the implementation of a solver, but can also reveal completeness and performance issues. Our experimental evaluation shows that our approach effectively finds errors in the implementation of widely-used SMT solvers and is also applicable to automata-based solvers. We synthesize sat formulas together with models and unsat formulas together with minimal unsat cores; having increasing complexity, our inputs facilitate error localization and debugging. We also showed how our approach can be extended to other theories and their combinations, how it can generate formulas with regular expressions, and how it can be used for testing optimization (MAX-SMT) solvers.

4

IDENTIFYING OVERLY RESTRICTIVE MATCHING PATTERNS IN SMT-BASED PROGRAM VERIFIERS

In this chapter, we present our technique for automatically identifying soundness issues in the axiomatizations produced by SMT-based program verifiers and for helping the developers reduce their incompleteness. We show that both problems occur when E-matching returns *unknown* for the SMT encoding of an axiomatization or of a proof obligation. As a solution, we propose a novel algorithm that synthesizes triggering terms and enables E-matching to refute the input formulas.

4.1 INTRODUCTION

Proof obligations frequently contain universal quantifiers, in the specification and to encode the semantics of the programming language. Most deductive verifiers [3, 7, 15, 45, 66, 106, 163] rely on SMT solvers to discharge their proof obligations via *E-matching* [63]. This SMT algorithm requires syntactic matching patterns of ground terms (called *patterns* in the following), to control the instantiations of the quantifiers. For example, the pattern $\{\mathtt{f}(x,y)\}$ in the formula $\forall x\colon \text{Int}, y\colon \text{Int} :: \{\mathtt{f}(x,y)\}\ (x = y) \wedge \neg \mathtt{f}(x,y)$ instructs the solver to instantiate the quantifier *only* when it finds a *triggering term* that matches the pattern, e.g., $\mathtt{f}(7,z)$, where `f` is an *uninterpreted* function and z is a free variable.

The patterns can be written manually or inferred automatically by the solver or the verifier. However, devising them is challenging [107, 122]. Too permissive patterns may lead to unnecessary instantiations that slow down verification or even cause non-termination (if each instantiation produces a new triggering term, in a so-called matching loop [63]). Overly restrictive patterns may prevent the instantiations needed to complete a proof; they cause two major problems in program verification: *incompleteness* and *undetected unsoundness*.

Incompleteness. Overly restrictive patterns may cause spurious verification errors when the proof of *valid* proof obligations fails. Figure 4.1 illustrates this case. The integer `x` represents the address of a node, and the uninterpreted functions `len` and `nxt` encode operations on linked lists. The axiom defines `len`: its result is positive, the last node points to itself, and any added node increases the length of the list by one. The assertion directly follows from the axiom, but the proof fails, as the proof obligation generated by the verifier for the `assert` statement does not contain any triggering term that matches the pattern `{len(nxt(x))}`. Thus, the axiom does not get instantiated. However, realistic proof obligations often contain *hundreds* of quantifiers [168], which makes the manual identification of missing triggering terms extremely difficult.

```
function len(x: int): int;
function nxt(x: int): int;

axiom (forall x: int :: {len(nxt(x))}
        len(x) > 0 && (nxt(x) == x ==> len(x) == 1) &&
        (nxt(x) != x ==> len(x) == len(nxt(x)) + 1));

procedure trivial() { assert len(7) > 0; }
```

FIGURE 4.1: Example (written in Boogie [14]) that leads to a spurious verification error. The assertion follows from the axiom, but the axiom does not get instantiated without the triggering term `len(nxt(7))`.

Unsoundness. Most of the universal quantifiers in proof obligations appear in axioms over uninterpreted functions (to encode type information, heap models, datatypes, etc.). To obtain sound results, these axioms must be consistent (i.e., satisfiable); otherwise, all the proof obligations hold trivially. Consistency can be proved once and for all by showing the existence of a model that satisfies all the axioms, as part of the soundness proof of the verification technique. However, this solution is difficult to apply for those verifiers that generate the axioms *dynamically*, depending on the program to be verified. Proving consistency then requires verifying the algorithm that generates the axioms for all possible inputs, and needs to consider many subtle issues [61, 108, 138].

A more practical approach is to check if the axioms generated for a given program are consistent. However, this check also depends on triggering: the SMT solver may fail to prove unsat if the triggering terms needed to instantiate the contradictory axioms are missing. The unsoundness can thus remain undetected.

For example, Dafny's [106] sequence axiomatization from June 2008 contained an inconsistency found only over a year later. A fragment of this axiomatization is

$$
\begin{aligned}
F_0\colon\quad & \forall t_0 \colon \mathrm{V} :: \{\mathtt{Type}(t_0)\}\ t_0 = \mathtt{ElemType}(\mathtt{Type}(t_0)) \\
F_1\colon\quad & \forall t_1 \colon \mathrm{V} :: \{\mathtt{Empty}(t_1)\}\ \mathtt{typ}(\mathtt{Empty}(t_1)) = \mathtt{Type}(t_1) \\
F_2\colon\quad & \forall s_2 \colon \mathrm{U}, i_2 \colon \mathrm{Int}, v_2 \colon \mathrm{U}, l_2 \colon \mathrm{Int} :: \{\mathtt{Build}(s_2, i_2, v_2, l_2)\} \\
& \qquad \mathtt{typ}(\mathtt{Build}(s_2, i_2, v_2, l_2)) = \mathtt{Type}(\mathtt{typ}(v_2)) \\
F_3\colon\quad & \forall s_3 \colon \mathrm{U} :: \{\mathtt{Len}(s_3)\}\ \neg(\mathtt{typ}(s_3) = \mathtt{Type}(\mathtt{ElemType}(\mathtt{typ}(s_3)))) \vee (0 \leq \mathtt{Len}(s_3)) \\
F_4\colon\quad & \forall s_4 \colon \mathrm{U}, i_4 \colon \mathrm{Int}, v_4 \colon \mathrm{U}, l_4 \colon \mathrm{Int} :: \{\mathtt{Len}(\mathtt{Build}(s_4, i_4, v_4, l_4))\} \\
& \qquad \neg(\mathtt{typ}(s_4) = \mathtt{Type}(\mathtt{typ}(v_4))) \vee (\mathtt{Len}(\mathtt{Build}(s_4, i_4, v_4, l_4)) = l_4)
\end{aligned}
$$

FIGURE 4.2: Fragment of an old version of Dafny's [106] sequence axiomatization. U and V are uninterpreted types. All the named functions are uninterpreted. To improve readability, we use mathematical notation throughout this chapter instead of SMT-LIB syntax [18].

shown in Figure 4.2. It expresses that empty sequences and sequences obtained through the `Build` operation are well-typed (F_0–F_2), that the length of a type-correct sequence must be non-negative (F_3), and that `Build` constructs a new sequence of the required length (F_4). The intended behavior of `Build` is to update the element at index i_4 in sequence s_4 to v_4. However, since there are no constraints on the parameter l_4, `Build` can be used with a negative length, leading to a contradiction with F_3. This unsoundness cannot be detected by checking the satisfiability of the formula $F_0 \wedge \ldots \wedge F_4$, as no axiom gets instantiated.

This work. For SMT-based deductive verifiers, discharging proof obligations and revealing inconsistencies in axiomatizations require the solver to prove *unsat* via E-matching. (Verification techniques based on proof assistants are out of scope.) Given an SMT formula for which E-matching yields *unknown* due to insufficient quantifier instantiations, our technique generates suitable triggering terms that allow the solver to complete the unsatisfiability proof. These terms enable tool users and developers to understand and remedy the revealed completeness or soundness issue. Since the SMT encodings of different input programs and their specifications typically share axiomatizations or parts of the verification condition that encode the semantics of the programming language, fixing such issues benefits the verification of many or even all future runs of the verifier.

Fixing the incompleteness. For Figure 4.1, our technique finds the triggering term `len(nxt(7))`, which allows one to fix the incompleteness. Tool *users* (who cannot change the axioms) can add the triggering term to the program. For example, adding the lines `var t: int; t := len(nxt(7));` before the assertion has no effect on the execution of the program, but triggers the instantiation of the axiom. Tool *developers* can devise less restrictive patterns; e.g., they can move the conjunct `len(x) > 0` to a separate axiom with the pattern `{len(x)}` (simply changing the axiom's pattern to `{len(x)}` would cause matching loops). Alternatively, they can use this information to adapt the encoding, to emit additional triggering terms enforcing certain instantiations [86, 107].

Fixing the unsoundness. In Figure 4.2, our synthesized triggering term $\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ (for a fresh value v) is sufficient to detect the unsoundness (see Section 4.2). Tool *users* can, based on this triggering term, report bugs in the implementation of the program verifier, while tool *developers* can add a precondition to F_4, which prevents the construction of sequences with negative lengths.

Soundness modulo patterns. Figure 4.3 illustrates another scenario: Boogie's [14] map axiomatization is inconsistent by design at the SMT level [109]; since F_2 states that storing a key-value pair into a map results in a new map with a potentially *different* type, one can prove that two *different* types (e.g., Boolean and Int) are equal in SMT. However, this behavior cannot be exposed from Boogie, as the type system prevents the required instantiations. Thus, it does not affect Boogie's soundness.

It is nevertheless important to detect it because it could surface if Boogie was extended to support quantifier instantiation algorithms that are not based on E-matching (such as MBQI [78]) or first-order provers (which also do not consider

$$
\begin{aligned}
F_0\colon\quad & \forall kt_0\colon \mathrm{V}, vt_0\colon \mathrm{V} :: \{\texttt{Type}(kt_0, vt_0)\}\ \texttt{ValTypeInv}(\texttt{Type}(kt_0, vt_0)) = vt_0\\
F_1\colon\quad & \forall m_1\colon \mathrm{U}, k_1\colon \mathrm{U}, v_1\colon \mathrm{U} :: \{\texttt{Select}(m_1, k_1, v_1)\}\\
& \qquad \texttt{typ}(\texttt{Select}(m_1, k_1, v_1)) = \texttt{ValTypeInv}(\texttt{typ}(m_1))\\
F_2\colon\quad & \forall m_2\colon \mathrm{U}, k_2\colon \mathrm{U}, x_2\colon \mathrm{U}, v_2\colon \mathrm{U} :: \{\texttt{Store}(m_2, k_2, x_2, v_2)\}\\
& \qquad \texttt{typ}(\texttt{Store}(m_2, k_2, x_2, v_2)) = \texttt{Type}(\texttt{typ}(k_2), \texttt{typ}(v_2))\\
F_3\colon\quad & \forall m_3\colon \mathrm{U}, k_3\colon \mathrm{U}, x_3\colon \mathrm{U}, v_3\colon \mathrm{U}, k_3'\colon \mathrm{U}, v_3'\colon \mathrm{U} :: \{\texttt{Select}(\texttt{Store}(m_3, k_3, x_3, v_3), k_3', v_3')\}\\
& \qquad (k_3 = k_3') \vee (\texttt{Select}(\texttt{Store}(m_3, k_3, x_3, v_3), k_3', v_3') = \texttt{Select}(m_3, k_3', v_3'))
\end{aligned}
$$

FIGURE 4.3: Fragment of Boogie's [14] map axiomatization, sound only modulo patterns. U and V are uninterpreted types. All the named functions are uninterpreted.

patterns). They could *unsoundly* classify an *in*valid Boogie program that uses this map axiomatization as valid. Since the verifier proves the validity of the verification condition by showing that its negation is unsatisfiable, if the refutation algorithm yields *unsat*, the verifier concludes that the program fulfills its specification. This is the case when checking the axioms from Figure 4.3 with MBQI: the formula $F_0 \wedge \ldots \wedge F_3$ is equivalent to `false`, so *any* (even invalid) Boogie program whose SMT encoding contains the axioms F_0–F_3 is reported as valid.

This example shows that the problems tackled in our work cannot be solved by switching to alternative instantiation strategies. First, these are not the preferred choices of most modern verifiers [3, 7, 15, 45, 66, 106, 163], and are, thus, unlikely to outperform E-matching. Second, they may produce unsound results for those verifiers designed for E-matching with axiomatizations sound *only modulo patterns*.

Contributions. This chapter makes the following technical contributions:

1. We present the first automated technique that allows users and developers of SMT-based program verifiers to detect *completeness* issues and *soundness* problems in their axiomatizations. Moreover, our approach helps them devise better triggering strategies for *all* future runs of their tool with E-matching.

2. We developed a novel algorithm for synthesizing the triggering terms necessary to complete unsatisfiability proofs using E-matching. Since quantifier instantiation is undecidable for first-order formulas over uninterpreted functions, our algorithm might not terminate. However, all identified triggering terms are sufficient to complete the proof, i.e., there are no false positives.

3. We evaluated our technique on benchmarks with known triggering problems from four program verifiers. Our experimental results show that it successfully synthesized the missing triggering terms in 65.6% of the cases and can significantly reduce the human effort in localizing and fixing the errors.

Outline. The rest of the chapter is organized as follows: Section 4.2 presents background information on E-matching. Section 4.3 gives an overview of our technique;

the details follow in Section 4.4. In Section 4.5, we present our experimental results, in Section 4.6 we describe various optimizations which allow our algorithm to scale to real-world inputs, and in Section 4.7 we explain its limitations. In Section 4.8 we briefly describe an alternative approach for synthesizing triggering terms starting from unsatisfiability proofs. We discuss related work in Section 4.9 and conclude in Section 4.10.

4.2 BACKGROUND: E-MATCHING

In this section, we discuss the E-matching-related terminology used in this chapter and explain how this quantifier-instantiation algorithm works on an example.

Patterns vs. triggering terms. Patterns are syntactic hints attached to quantifiers which instruct the SMT solver when to perform an instantiation. In Figure 4.2, the quantified formula F_3 will be instantiated only when a *triggering term* that matches the *pattern* $\{\texttt{Len}(s_3)\}$ is encountered during the SMT run (i.e., the triggering term is present in the quantifier-free part of the input formula or is obtained by the solver from the body of a previously-instantiated quantifier). Patterns are matched *modulo equalities*, that is, F_4, which has the pattern $\{\texttt{Len}(\texttt{Build}(s_4, i_4, v_4, l_4))\}$, will be instantiated also when the solver is provided the triggering term $\texttt{Len}(s)$ and it knows that $s = \texttt{Build}(s_4, i_4, v_4, l_4)$ holds for some $s_4\colon \text{U}, i_4\colon \text{Int}, v_4\colon \text{U}, l_4\colon \text{Int}$. However, our algorithm does not generate such triggering terms, as it automatically substitutes s by the right hand side of the equality.

E-matching. We now illustrate how E-matching works on the example from Figure 4.2; in particular, we show how our synthesized triggering term $\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ helps the solver to prove unsat when added to the axiomatization (v is a fresh variable of type U). To keep the explanation concise, we omit unnecessary instantiations. First, the sub-terms $\texttt{Empty}(\texttt{typ}(v))$ and $\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ trigger the instantation of F_1 and F_4, respectively. The solver obtains the body of the quantifiers for *these particular values*:

$$\begin{aligned} B_1\colon\ & \texttt{typ}(\texttt{Empty}(\texttt{typ}(v))) = \texttt{Type}(\texttt{typ}(v)) \\ B_4\colon\ & \neg(\texttt{typ}(\texttt{Empty}(\texttt{typ}(v))) = \texttt{Type}(\texttt{typ}(v))) \vee \\ & (\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)) = -1) \end{aligned}$$

As the first disjunct of B_4 evaluates to `false` (from B_1), the solver learns that the second disjunct must hold (i.e., the length must be -1); we abbreviate it as $L = -1$. The sub-terms $\texttt{Build}(\texttt{Empty}(\texttt{typ}(v))$ and $\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))$ of the synthesized triggering term lead to the instantiation of F_2 and F_3, respectively:

$$\begin{aligned} B_2\colon\ & \texttt{type}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)) = \texttt{Type}(\texttt{typ}(v)) \\ B_3\colon\ & \neg(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)) = \\ & \texttt{Type}(\texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))))) \vee \\ & (0 \leq \texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)))) \end{aligned}$$

$\texttt{Type}(\texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))))$ from B_3 triggers F_0:

$$B_0\colon\ \texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1))) = \texttt{ElemType}(\texttt{Type}(\texttt{ElemType}(\texttt{typ}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)))))$$

By equalizing the arguments of the outermost `ElemType` in B_0, the solver learns that the first disjunct of B_3 is `false`. The second disjunct must thus hold (i.e., the length should be positive); we abbreviate it as $0 \leq L$. Since $(L = -1) \wedge (0 \leq L) =$ `false`, the unsatisfiability proof succeeds.

4.3 OVERVIEW

Our goal is to synthesize missing triggering terms, i.e., concrete instantiations for (a small subset of) the quantified variables of an input formula `I`, which are necessary for the solver to prove its unsatisfiablity. Intuitively, these triggering terms include *counterexamples* to the satisfiability of `I` and can be obtained from a model of its negation. For example, $\texttt{I} = \forall n\colon \text{Int} :: n > 7$ is unsatisfiable, and a counterexample $n = 6$ is a model of its negation $\neg\texttt{I} = \exists n\colon \text{Int} :: n \leq 7$.

However, this idea does not apply to formulas over uninterpreted functions, which are common in proof obligations. The negation of $\texttt{I} = \exists \texttt{f}, \forall n\colon \text{Int} :: \texttt{f}(n, 7)$, where `f` is an uninterpreted function, is $\neg\texttt{I} = \forall \texttt{f}, \exists n\colon \text{Int} :: \neg\texttt{f}(n, 7)$. This is a second-order constraint (it quantifies over functions) and cannot be directly encoded in SMT. We thus take a different approach.

Let F be a second-order formula, in which universal quantifiers appear only in positive positions. We define its *approximation* as:

$$F_{\approx} = F[\exists \overline{\texttt{f}} \,/\, \forall \overline{\texttt{f}}] \tag{4.1}$$

where $\overline{\texttt{f}}$ are uninterpreted functions. The approximation considers only *one* interpretation, not *all* possible interpretations for each uninterpreted function.

We, therefore, construct a *candidate* triggering term from a model of $\neg\texttt{I}_{\approx}$ and check if it is sufficient to prove that `I` is unsatisfiable (due to the approximation, a model is no longer guaranteed to be a counterexample for the original formula).

The four main steps of our algorithm are depicted in Figure 4.4. The algorithm is stand-alone, i.e., not integrated into, nor dependent on any specific SMT solver. We illustrate it on the inconsistent axioms from Figure 4.5 (which we assume are part of a larger axiomatization). To show that $\texttt{I} = F_0 \wedge F_1 \wedge \ldots$ is unsatisfiable, the solver requires the triggering term $\texttt{f}(\texttt{g}(7))$. The corresponding instantiations of F_0 and F_1 generate contradictory constraints: $\texttt{f}(\texttt{g}(7)) \neq 7$ and $\texttt{f}(\texttt{g}(7)) = 7$. In the following, we explain how we obtain this triggering term systematically.

Step 1: Clustering. As typical proof obligations or axiomatizations contain hundreds of quantifiers, exploring combinations of triggering terms for all of them

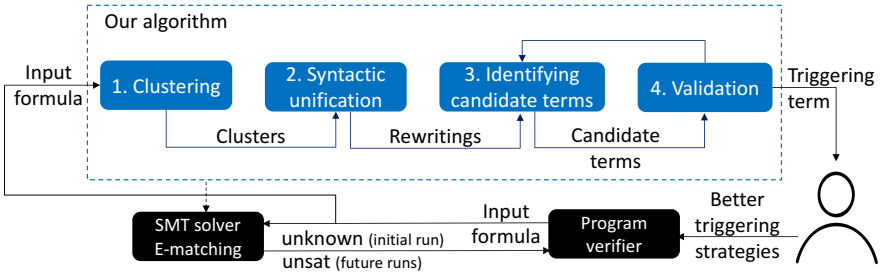

FIGURE 4.4: Main steps of our algorithm (represented as blue boxes), which helps the developers of SMT-based verifiers devise better triggering strategies (and enable E-matching to prove unsat). The arrows depict data.

does not scale. To prune the search space, we exploit the fact that `I` is unsatisfiable only if there exist instantiations of some (in the worst case all) of its *quantified* conjuncts F such that they produce contradictory constraints on some uninterpreted functions. (If there is a contradiction among the quantifier-free conjuncts, the solver will detect it directly.) We thus identify *clusters* C of formulas F that share function symbols and then process each cluster separately. In Figure 4.5, F_0 and F_1 share the function symbol `f`, so we build the cluster $C = F_0 \wedge F_1$.

Step 2: Syntactic unification. The formulas within clusters usually contain uninterpreted functions applied to *different* arguments (e.g., `f` is applied to x_0 in F_0 and to $\mathtt{g}(x_1)$ in F_1). We thus perform syntactic unification to identify *sharing constraints* on the quantified variables (which we call *rewritings* and denote their set by R) such that instantiations that satisfy these rewritings generate formulas with common terms (on which they might set contradictory constraints). F_0 and F_1 share the term $\mathtt{f}(\mathtt{g}(x_1))$ if we perform the rewritings $R = \{x_0 = \mathtt{g}(x_1)\}$.

Step 3: Identifying candidate triggering terms. The cluster $C = F_0 \wedge F_1$ from step 1 contains a contradiction if there exists a formula F_i in C such that: (1) F_i is unsatisfiable by itself, or (2) F_i contradicts at least one other formula from C.

To address scenario (1), we ask a solver for a model of the formula $G = \neg C_{\approx}$, where $\neg C_{\approx}$ is defined in (4.1). After Skolemization, G is quantifier-free, so the solver is generally able to provide a model, if one exists. We then obtain a candi-

$$F_0: \quad \forall x_0 : \mathrm{Int} :: \{\mathtt{f}(x_0)\} \;\; \mathtt{f}(x_0) \neq 7$$
$$F_1: \quad \forall x_1 : \mathrm{Int} :: \{\mathtt{f}(\mathtt{g}(x_1))\} \;\; \mathtt{f}(\mathtt{g}(x_1)) = x_1$$

FIGURE 4.5: Formulas that set contradictory constraints on the uninterpreted function `f`. Synthesizing the triggering term $\mathtt{dummy}(\mathtt{f}(\mathtt{g}(7)))$ requires theory reasoning and syntactic unification. `dummy` is a fresh uninterpreted function (see Step 4).

date triggering term by substituting the quantified variables from the patterns of the formulas in C with their corresponding values from the model.

However, scenario (1) is not sufficient to expose the contradiction from Figure 4.5, since both F_0 and F_1 are individually satisfiable. Our algorithm thus also derives *stronger* G formulas corresponding to scenario (2). That is, it will next consider the case where F_0 contradicts F_1, whose encoding into first-order logic is: $\neg F_{0\approx} \wedge F_1 \wedge \bigwedge R$, where R is the set of rewritings identified in step 2, used to connect the quantified variables. This formula is universally-quantified (since F_1 is), so the solver cannot prove its satisfiability and generate models. We solve this issue by requiring F_0 to contradict the *instantiation* of F_1, which is a weaker constraint.

Let F be an arbitrary formula, with universal quantifiers only in positive positions. We define its *instantiation* as:

$$F_{Inst} = F[\exists \overline{\mathtt{x}} \,/\, \forall \overline{\mathtt{x}}] \tag{4.2}$$

where $\overline{\mathtt{x}}$ are variables. Then $G = \neg F_{0\approx} \wedge F_{1\,Inst} \wedge \bigwedge R$ is equivalent to $(\mathtt{f}(x_0) = 7) \wedge (\mathtt{f}(\mathtt{g}(x_1)) = x_1) \wedge (x_0 = \mathtt{g}(x_1))$. (To simplify the notation, here and in the following formulas, we omit existential quantifiers.) All its models set x_1 to 7. Substituting x_0 by $\mathtt{g}(x_1)$ (according to R) and x_1 by 7 (its value from the model) in the patterns of F_0 and F_1 yields the candidate triggering term $\mathtt{f}(\mathtt{g}(7))$.

Step 4: Validation. Once we have found a candidate triggering term, we add it to the original formula `I` (wrapped in a fresh uninterpreted function `dummy`, to make it available to E-matching, but not affect the input's satisfiability) and check if the solver can prove unsat. If so, our algorithm terminates successfully and reports the synthesized triggering term (after a minimization step that removes unnecessary sub-terms); otherwise, we go back to step 3 to obtain another candidate. In our example, the triggering term $\mathtt{dummy}(\mathtt{f}(\mathtt{g}(7)))$ is sufficient to complete the proof.

4.4 SYNTHESIZING TRIGGERING TERMS

In the following, we present our algorithm for synthesizing triggering terms required by E-matching to return unsat: in Section 4.4.1 we define the input formulas and in Section 4.4.2 we explain the details of the algorithm. Its extensions follow in Section 4.4.3. We illustrate the algorithm on additional examples in Section 4.4.4.

4.4.1 *Input formula*

To simplify our algorithm, we pre-process the inputs (i.e., the proof obligations or the axioms of a verifier): we Skolemize existential quantifiers and transform all propositional formulas into *negation normal form* (NNF), where negation is applied only to literals and the only logical connectives are conjunction and disjunction; we also apply the distributivity of disjunction over conjunction and split conjunctions into separate formulas. These steps preserve satisfiability and the semantics of patterns (Section 4.6 addresses scalability issues). The resulting formulas follow the

$$\begin{array}{llll} \texttt{I} & ::= & F\ (\wedge\ F)^* & B \quad ::= \quad D\ (\vee\ D)^* \\ F & ::= & B \mid \forall \overline{x} :: \{P(\overline{x})\}\ B & D \quad ::= \quad L \mid \neg L \mid \forall \overline{x} :: \{P(\overline{x})\}\ F \end{array}$$

FIGURE 4.6: Grammar of input formulas `I`. Inputs are conjunctions of formulas F, which are (typically quantified) disjunctions of literals (L or $\neg L$) or nested quantified formulas. Each quantifier has a pattern P. $\overline{x}$ is a (non-empty) list of variables.

grammar from Figure 4.6. Literals L may include interpreted and uninterpreted functions, variables and constants. Free variables are nullary functions. Quantified variables can have interpreted or uninterpreted types, and the pre-processing ensures that their names are globally unique. We assume that each quantifier is equipped with a pattern P (if none is provided, we run the solver to infer one). Patterns are combinations of *uninterpreted* functions and must mention *all* quantified variables. Since there are no existential quantifiers after Skolemization, in the rest of this chapter we use the term *quantifier* to denote *universal quantifiers*.

4.4.2 *Algorithm*

The pseudo-code of our algorithm is given in Algorithm 4.1. It takes as input an SMT formula `I` (defined in Figure 4.6), which we treat in a slight abuse of notation as both a formula and a set of conjuncts. Three other parameters allow us to customize the search strategy and are discussed later. The algorithm yields a triggering term that enables the unsat proof, or `None` if no term was found. We assume here that `I` contains no nested quantifiers and present those later.

The algorithm iterates over each *quantified* conjunct F of `I` (Algorithm 4.1, line 3) and checks if F is individually unsatisfiable (for `depth = 0`). For complex proofs, this is usually not sufficient, as `I` is typically inconsistent due to *a combination* of conjuncts ($F_0 \wedge F_1$ in Figure 4.5). In such cases, the algorithm proceeds as follows:

Step 1: Clustering. It constructs clusters of formulas similar to F (Algorithm 4.2, line 4), based on their *Jaccard similarity index*. Let F_i and F_j be two arbitrary formulas, and S_i and S_j their respective sets of uninterpreted function symbols (from their bodies and the patterns). The Jaccard similarity index is defined as:

$J(F_i, F_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$ (the number of common uninterpreted functions divided by the total number). For Figure 4.5, $S_0 = \{\texttt{f}\}$, $S_1 = \{\texttt{f}, \texttt{g}\}$, $J(F_0, F_1) = \frac{|\{\texttt{f}\}|}{|\{\texttt{f},\texttt{g}\}|} = 0.5$.

Our algorithm explores the search space by iteratively expanding clusters to include transitively-similar formulas up to a maximum depth (parameter δ in Algorithm 4.1). For two formulas $F_i, F_j \in \texttt{I}$, we define the similarity function as:

$$\texttt{sim}^{\delta}_{\texttt{I}}(F_i, F_j, \sigma) = \begin{cases} J(F_i, F_j) \geq \sigma, & \delta = 1 \\ \exists F_k : \texttt{sim}^{\delta-1}_{\texttt{I}\setminus\{F_i\}}(F_i, F_k, \sigma) \text{ and } J(F_k, F_j) \geq \sigma, & \delta > 1 \end{cases}$$

where $\sigma \in [0, 1]$ is a similarity threshold used to parameterize our algorithm.

Algorithm 4.1: Our algorithm for synthesizing triggering terms that enable unsatisfiability proofs. We assume here that all quantified variables are globally unique and `I` does not contain nested quantifiers. The auxiliary procedures `clustersRewritings` and `candidateTerm` are presented in Algorithm 4.2 and Algorithm 4.3, respectively.

Arguments : `I` — input formula, also treated as set of conjuncts
σ — similarity threshold for clustering
δ — maximum depth for clustering
μ — maximum number of different models
Result: The synthesized triggering term or None, if no term was found

```
1  Procedure synthesizeTriggeringTerm
2      foreach depth ∈ {0, ..., δ} do
3          foreach F ∈ I | F is ∀x̄ :: F′ do
4              foreach (C, R) ∈ clustersRewritings(I, F, σ, depth) do      // steps 1, 2
5                  Inst ⟵ {}
6                  foreach f ∈ C | f is ∀x̄ :: D_0 ∨ ... ∨ D_n or D_0 ∨ ... ∨ D_n do
7                      Inst[f] ⟵ {(⋀_{0≤j<k} ¬D_j) ∧ D_k | 0 ≤ k ≤ n}
8                  Inst[F] ⟵ {¬F′}
9                  foreach H ∈ ⨉{Inst[f] | f ∈ {F} ∪ C} do      // Cartesian product
10                     G ⟵ ⋀H ∧ ⋀R
11                     foreach m ∈ {0, ..., μ − 1} do
12                         resG, model ⟵ checkSat(G)
13                         if resG ≠ SAT then
14                             break      // no models if G is not SAT
15                         T ⟵ candidateTerm({F} ∪ C, R, model)      // step 3
16                         resI, _ ⟵ checkSat(I ∧ T)      // step 4
17                         if resI = UNSAT then
18                             return minimized(T)      // success
19                         G ⟵ G ∧ ¬model      // avoid this model next iteration
20     return None
```

The initial cluster (`depth` $= 1$) includes all the conjuncts of `I` that are *directly* similar to F. Each subsequent iteration adds the conjuncts that are directly similar to an element of the cluster from the previous iteration, that is, *transitively* similar to F. This search strategy allows us to gradually strengthen the formulas G (used to synthesize candidate terms in step 3) without overly constraining them (an over-constrained formula is unsatisfiable, and has no models).

Step 2: Syntactic unification. Next (Algorithm 4.2, line 8) we identify *rewritings*, i.e., constraints under which two similar *quantified* formulas share terms. (Section 4.4.4 presents the quantifier-free case.) We obtain the rewritings by performing

Algorithm 4.2: Auxiliary procedure for Algorithm 4.1, which identifies clusters of formulas similar to F and their rewritings. `sim` is defined in text (step 1). `unify` is a first-order unification algorithm (not shown); it returns a set of rewritings with restricted shapes, defined in text (step 2).

Arguments : `I` — input formula, also treated as set of conjuncts
F — quantified conjunct of `I`, i.e., $F \in \mathtt{I} \mid F$ is $\forall \overline{x} :: F'$
σ — similarity threshold for clustering
depth — current depth for clustering

Result: A set of pairs, consisting of clusters and their corresponding rewritings

Procedure clustersRewritings
  if depth = 0 **then**
    return $\{(\varnothing, \varnothing)\}$
  simFormulas $\longleftarrow \{f \mid f \in \mathtt{I} \setminus \{F\}$ and $\mathtt{sim}_{\mathtt{I}}^{\mathsf{depth}}(\mathrm{F}, \mathrm{f}, \sigma)\}$ // step 1
  rewritings $\longleftarrow \{\}$
  foreach $f \in$ simFormulas **do**
    rws $\longleftarrow$ unify(F, f) // step 2
    if rws $= \varnothing$ and (f is $\forall \overline{x} :: D_0 \vee \ldots \vee D_n$) **then**
      simFormulas $\longleftarrow$ simFormulas $\setminus \{f\}$
    rewritings$[f] \longleftarrow$ rws
  return $\{(C, R) \mid C \subseteq$ simFormulas and $(\forall r \in R, \exists f \in C : r \in$ rewritings$[f])$
    and $(\forall x \in \mathtt{qvars}(C) : |\{r \mid r \in R$ and $x = \mathtt{lhs}(r)\}| \leq 1)\}$

a *simplified* form of *syntactic term unification*, which reduces their number to a practical size. Our rewritings are *directed equalities*. For two formulas F_i and F_j and an uninterpreted function `f` they have one of the following two shapes:

(1) $x_i = rhs_j$, where x_i is a quantified variable of F_i, rhs_j are terms from F_j defined below, F_i contains a term $\mathtt{f}(x_i)$ and F_j contains a term $\mathtt{f}(rhs_j)$,

(2) $x_j = rhs_i$, where x_j is a quantified variable of F_j, rhs_i are terms from F_i defined below, F_j contains a term $\mathtt{f}(x_j)$ and F_i contains a term $\mathtt{f}(rhs_i)$,

where rhs_k is a constant c_k, a quantified variable x_k, or a composite function $(\mathtt{f} \circ \mathtt{g}_0 \circ \ldots \circ \mathtt{g}_n)(\overline{c_k}, \overline{x_k})$ occurring in the formula F_k and $\mathtt{g}_0, \ldots, \mathtt{g}_n$ are arbitrary (interpreted or uninterpreted) functions. That is, we determine the *most general unifier* [10] only for those terms that have uninterpreted functions as the outermost functions and quantified variables as arguments. The unification algorithm is standard (except for the restricted shapes), so it is not shown explicitly.

In Figure 4.5, F_1 is similar to F_0 for any $\sigma \leq 0.5$. We then compute the rewritings for all the quantified variables of F_0 that appear in its body as arguments to some *common* uninterpreted functions (in this case, only x_0). Unifying the terms $\mathtt{f}(x_0)$ and $\mathtt{f}(\mathtt{g}(x_1))$ generates the rewriting $x_0 = \mathtt{g}(x_1)$, which has shape (1).

Since a term may appear more than once in F, or F unifies with multiple similar formulas through the same quantified variable, we can obtain *alternative rewritings*

$$
\begin{aligned}
F_0\colon &\quad \forall x_0 \colon \mathrm{Int} :: \{\mathtt{f}(x_0)\}\ \mathtt{f}(x_0) = 6 \\
F_1\colon &\quad \forall x_1 \colon \mathrm{Int} :: \{\mathtt{f}(x_1)\}\ \mathtt{f}(x_1) = 7 \\
F_2\colon &\quad \forall x_2 \colon \mathrm{Int} :: \{\mathtt{f}(x_2)\}\ \mathtt{f}(x_2) = 8
\end{aligned}
$$

FIGURE 4.7: Formulas that set contradictory constraints on the uninterpreted function `f`. Synthesizing the triggering term `dummy(f(0))` requires clusters of similar formulas with alternative rewritings.

for a quantified variable. In such cases, we either duplicate or split the cluster, such that in each cluster-rewriting pair, each quantified variable is rewritten at most once (see Algorithm 4.2, line 12). In Figure 4.7, both F_1 and F_2 are similar to F_0 (all three formulas share the uninterpreted symbol `f`). Since the unification produces alternative rewritings for x_0 ($x_0 = x_1$ and $x_0 = x_2$), the procedure `clustersRewritings` returns the pairs $\{(\{F_1\}, \{x_0 = x_1\}), (\{F_2\}, \{x_0 = x_2\})\}$.

Step 3: Identifying candidate terms. From the clusters and the rewritings (identified before), we then derive *quantifier-free* formulas G (Algorithm 4.1, line 10), and, if they are satisfiable, construct the candidate triggering terms from their models (Algorithm 4.1, line 15). Each formula G consists of: (1) $\neg F_{\approx}$ (defined in (4.1), which is equivalent to $\neg F'$, since F has the shape $\forall \overline{x} :: F'$ from Algorithm 4.1, line 3), (2) the *instantiations* (defined in (4.2)) of all the similar formulas from the cluster, and (3) the corresponding rewritings R. (Since we assume that all the quantified variables are globally unique, we do not perform variable renaming when computing the instantiations).

If a similar formula has multiple disjuncts D_k, the SMT solver may use short-circuiting semantics when generating the model for G. That is, if it can find a model that satisfies the first disjunct, it may not consider the remaining ones. To obtain more diverse models, we synthesize formulas that *cover* each disjunct, i.e., make sure that it evaluates to `true` at last once. We thus compute *multiple instantiations* of each similar formula, of the form: $(\bigwedge_{0 \leq j < k} \neg D_j) \wedge D_k, \forall k \colon 0 \leq k \leq n$ (see Algorithm 4.1, line 7). To consider all the combinations of disjuncts, we derive the formula G from the Cartesian product of the instantiations (Algorithm 4.1, line 9). (For presentation purposes, we also store $\neg F'$ in the instantiations map (Algorithm 4.1, line 8), even if it does *not* represent the instantiation of F.)

$$
\begin{aligned}
F_0\colon &\quad \forall x_0 \colon \mathrm{Int} :: \{\mathtt{f}(x_0)\}\ \neg(x_0 > -1) \vee \mathtt{f}(x_0) > 7 \\
F_1\colon &\quad \forall x_1 \colon \mathrm{Int} :: \{\mathtt{f}(x_1)\}\ \neg(x_1 < 1) \vee \mathtt{f}(x_1) = 6
\end{aligned}
$$

FIGURE 4.8: Formulas that set contradictory constraints on the uninterpreted function `f`. Synthesizing the triggering term `dummy(f(0))` requires instantiations that cover all the disjuncts.

Algorithm 4.3: Auxiliary procedure for Algorithm 4.1, which constructs a triggering term from the given cluster, rewritings, and SMT model. `dummy` is a fresh function symbol, which conveys no information about the truth value of the candidate term; thus conjoining it to the input preserves (un)satisfiability.

Arguments : C — set of formulas in the cluster
R — set of rewritings for the cluster
`model` — SMT model, mapping variables to values
Result: A triggering term with no semantic information

```
1 Procedure candidateTerm
2     P_0, ..., P_k ⟵ patterns(C)
3     while R ≠ ∅ do
4         choose and remove r ⟵ (x = rhs) from R
5         P_0, ..., P_k ⟵ (P_0, ..., P_k)[rhs/x]
6         R ⟵ R[rhs/x]
7     foreach x ∈ qvars(C) do
8         P_0, ..., P_k ⟵ (P_0, ..., P_k)[model(x)/x]
9     return "dummy" + "(" + P_0, ..., P_k + ")"
```

In Figure 4.8, F_1 is similar to F_0 and $R = \{x_0 = x_1\}$. F_1 has two disjuncts and thus two possible instantiations: $\mathsf{Inst}[F_1] = \{x_1 \geq 1, (x_1 < 1) \wedge (\mathtt{f}(x_1) = 6)\}$. The formula $G = (x_0 > -1) \wedge (\mathtt{f}(x_0) \leq 7) \wedge (x_1 \geq 1) \wedge (x_0 = x_1)$ for the first instantiation is satisfiable, but none of the values the solver can assign to x_0 (which are all greater or equal to 1) are sufficient for the unsatisfiability proof to succeed. The second instantiation adds additional constraints: instead of $x_1 \geq 1$, it requires $(x_1 < 1) \wedge (\mathtt{f}(x_1) = 6)$. The resulting G formula has a unique solution for x_0, namely 0, and the triggering term $\mathtt{f}(0)$ is sufficient to prove unsat.

The procedure candidateTerm in Algorithm 4.3 synthesizes a candidate triggering term T from the models of G and the rewritings R. We first collect all the patterns of the formulas from the cluster C (Algorithm 4.3, line 2), i.e., of F and of its similar conjuncts (see Algorithm 4.1, line 15). Then, we *apply* the rewritings, in an arbitrary order (Algorithm 4.3, lines 3–6). That is, we substitute the quantified variable x from the left-hand side of the rewriting with the right-hand side term *rhs* and propagate this substitution to the remaining rewritings. This step allows us to include in the synthesized triggering terms additional information, which cannot be provided by the solver. Then (Algorithm 4.3, lines 7–8) we substitute the remaining variables with their *constant* values from the model (i.e., constants for built-in types, and fresh, unconstrained variables for uninterpreted types). For interpreted, user-defined types (such as a type `IList` for representing a `List` of `Int`, where `List` and `Int` are both interpreted types), the solver generates constants for each type component, or a sequence of operations required to construct them. For instance, `insert(0, nil)` (i.e., a singleton list containing the constant 0) is a possible model provided by the SMT solver Z3 [187] for a variable of type `IList`. The

$$F\colon \quad \forall x\colon \text{Int}, y\colon \text{Int} :: \{\texttt{\_div}(x,y)\}\ \texttt{\_div}(x,y) = x/y$$

FIGURE 4.9: Inconsistent axiom from F* [162]. `_div`: Int × Int → Int is an uninterpreted function. Synthesizing the triggering term $\texttt{dummy}(\texttt{\_div}(1,2))$ requires diverse models.

resulting triggering term is wrapped in an application to a fresh, uninterpreted function `dummy` to ensure that conjoining it to `I` does not change `I`'s satisfiability.

Step 4: Validation. We validate the candidate triggering term T by checking if $\texttt{I} \wedge T$ is unsatisfiable, i.e., if these particular interpretations for the uninterpreted functions generalize to all interpretations (Algorithm 4.1, line 16). If this is the case then we return the *minimized* triggering term (Algorithm 4.1, line 18). The `dummy` function has multiple arguments, each of them corresponding to one pattern from the cluster (Algorithm 4.3, line 9). This is an over-approximation of the required triggering terms (once instantiated, the formulas may trigger each other), so `minimized` removes redundant (sub-)terms. If T does not validate, we re-iterate its construction up to a bound μ and strengthen the formula G to obtain a different model (Algorithm 4.3, lines 19 and 11). The parameter μ allows us to deal with other sources of incompleteness, as we explain next.

Let us consider the formula from Figure 4.9, which was part of an axiomatization with 2 495 axioms. F axiomatizes the uninterpreted function `_div`: Int × Int → Int and is inconsistent, because there exist two integers whose real division ("/") is not an integer. The model produced by Z3 for the formula $G = \neg F'$ is $x = -1, y = 0$. $-1/0$ is defined ("/" is a total function [18]), but its result is not specified. Thus the solver cannot validate this model (i.e., it returns *unknown*).

In such cases, we ask the solver for a different model. In Figure 4.9, if we simply exclude previous models, we can obtain a sequence of models with different values for the numerator, but with the same value (0) for the denominator. There are infinitely many such models, and all of them fail to validate for the same reason.

There are various heuristics one can employ to guide the solver's search for new models; our algorithm can be parameterized with different ones. In our experiments, we interpret the conjunct $\neg\texttt{model}$ from Algorithm 4.1, line 19 as $(\bigwedge_{x\in\overline{x}} x \neq \texttt{model}(x)) \wedge (\bigwedge_{x_i,x_j\in\overline{x},\, i\neq j,\, \texttt{model}(x_i)=\texttt{model}(x_j)} x_i \neq x_j)$. This allows us to synthesize the triggering term $\texttt{dummy}(\texttt{\_div}(1,2))$ and expose the inconsistency from Figure 4.9. The first component $(\bigwedge_{x\in\overline{x}} x \neq \texttt{model}(x))$ requires all the variables to have different values than before. This requirement may be too strong for some variables, but as we use only *soft* constraints, the solver may ignore some of them if it cannot generate a satisfying assignment. The second part $(\bigwedge_{x_i,x_j\in\overline{x},\, i\neq j,\, \texttt{model}(x_i)=\texttt{model}(x_j)} x_i \neq x_j)$ requires models from different *equivalence classes*, where an equivalence class includes all the variables that are equal in the model. For example, if the model is $x_0 = x, x_1 = x$, where x is a value of the corresponding type, then x_0 and x_1 belong to the same equivalence class. Considering equivalence classes is par-

ticularly important for variables of uninterpreted types; the solver cannot provide actual values for them, thus it assigns fresh, unconstrained variables. However, different fresh variables do not lead to diverse models.

Nested quantifiers. Our algorithm also supports nested quantifiers. Nested existential quantifiers in positive positions and nested universal quantifiers in negative positions are replaced in NNF by new, uninterpreted Skolem functions. Step 2 is also applicable to them: Skolem functions with arguments (the quantified variables from the outer scope) are unified as regular uninterpreted functions; they can also appear as *rhs* in a rewriting, but not as the left-hand side (we do not perform higher-order unification). In such cases, the result is imprecise: the unification of $\texttt{f}(x_0, \texttt{skolem}())$ and $\texttt{f}(x_1, 1)$ produces only the rewriting $x_0 = x_1$.

After pre-processing, the conjunct F and the similar formulas may still contain *nested universal quantifiers*. F is always negated in G, thus it becomes, after Skolemization, quantifier-free. To ensure that G is also quantifier-free (and the solver can generate a model for it), we extend the algorithm to *recursively instantiate* similar formulas with nested quantifiers when computing the instantiations.

4.4.3 *Extensions*

Next, we describe various extensions of our algorithm that enable complex proofs.

Combining multiple candidate terms. In Algorithm 4.1, each candidate term is validated separately. To enable proofs that require *multiple instantiations* of the *same* formula, we developed an extension that validates multiple triggering terms at the same time. In such cases, the algorithm returns a *set of terms* that are necessary and sufficient to prove unsat. Figure 4.10 presents a simple example from SMT-COMP 2019 pending benchmarks [167]. (The files in this category are not guaranteed to comply with the SMT-LIB standard, but our benchmarks selection algorithm described in Section 4.5.2 checks this automatically.) The input $\texttt{I} = F_0 \wedge F_1$ is unsatisfiable, as there does not exist an interpretation for the uninterpreted function `U` that satisfies all the constraints: F_1 requires $\texttt{U}(\texttt{s})$ to be `true`; if F_0 is instantiated for $x_0 = \texttt{s}$, the solver learns that $\texttt{U}(\texttt{il})$ must be `true` as well; however, if $x_0 = \texttt{il}$, then $\texttt{U}(\texttt{il})$ must be `false`, which is a contradiction. Therefore, exposing the incon-

$$F_0\colon \quad \forall x_0 \colon \texttt{S} :: \{\texttt{f}(x_0)\}\ \neg\texttt{U}(x_0) \vee (\texttt{U}(\texttt{f}(x_0)) \wedge \texttt{f}(x_0) = \texttt{il} \wedge x_0 \neq \texttt{il})$$
$$F_1\colon \quad \texttt{U}(\texttt{s})$$

FIGURE 4.10: Benchmark from SMT-COMP 2019 [167]. The formulas set contradictory constraints on the uninterpreted function `U`. S is an uninterpreted type, `s` and `il` are user-defined constants of type S. Synthesizing the triggering term $\texttt{dummy}(\texttt{f}(\texttt{s}), \texttt{f}(\texttt{il}))$ requires multiple candidate terms. We use conjunctions here for simplicity, but our pre-processing applies distributivity of disjunction over conjunction and splits F_0 into three different formulas with unique names for the quantified variables.

$$F\colon \quad \forall x\colon \mathrm{Int} \;::\; \{\mathtt{g}(x)\}\ \mathtt{f}(x) \neq \mathtt{f}(7)$$

FIGURE 4.11: Formula that sets contradictory constraints on the uninterpreted function `f`. The uninterpreted function `g` is used only as a pattern (i.e., it does not appear in the body of F, see Section 4.4.4). Synthesizing the triggering term $\mathtt{dummy}(\mathtt{g}(7))$ requires unification across multiple instantiations.

sistency requires two instantiations of F_0, triggered by $\mathtt{f}(\mathtt{s})$ and $\mathtt{f}(\mathtt{i1})$, respectively. We generate both triggering terms, but in separate iterations (independently, both fail to validate). However, by validating them simultaneously (i.e., conjoining both of them to `I`, as arguments to the fresh function `dummy`), our algorithm identifies the required triggering term $T = \mathtt{dummy}(\mathtt{f}(\mathtt{s}), \mathtt{f}(\mathtt{i1}))$.

Unification across multiple instantiations. The clusters constructed by our algorithm are sets (see Algorithm 4.2, line 12), so they contain a formula at most once, even if it is similar to multiple other formulas from the cluster. We thus consider the rewritings for multiple instantiations of the same formula separately, in different iterations. To handle cases that require *multiple* (but boundedly many) *instantiations*, we extend the algorithm with a parameter Φ, which bounds the maximum frequency of a *quantified* conjunct within the formulas G. That is, it allows a similar quantified formula, as well as F itself, to be added to a cluster (now represented as a list) more than once (after performing variable renaming, to ensure that the names of the quantified variables are still globally unique). This results in an equisatisfiable formula for which our algorithm determines multiple triggering terms. Inputs whose unsatisfiability proofs require an *unbounded* number of instantiations typically contain a matching loop, thus we do not consider them here. Figure 4.11 presents an example, which consists of a single inconsistent formula F. Our regular algorithm from Algorithm 4.2 does not identify any rewritings. However, with this extension, F unifies with itself for any $\Phi > 1$, and one possible rewriting is $x' = 7$ (x' is a fresh variable representing the second instantiation of F). The corresponding triggering term, $T = \mathtt{dummy}(\mathtt{g}(7))$, allows E-matching to prove unsat.

Note that the uninterpreted function `g` is used only as a pattern. If the pattern were $\mathtt{f}(x)$, any triggering term $\mathtt{f}(c)$, where c is an integer constant, were sufficient to complete the proof: (1) for $c = 7$, the contradiction would have been exposed directly; (2) for $c \neq 7$, the term $\mathtt{f}(7)$, obtained from the first instantiation of F, would have triggered its second instantiation and case (1) would have then applied.

Type-based constraints. The rewritings of the form $x_i = x_j$ can be too imprecise (especially for quantified variables of uninterpreted types), as they do not constrain the *rhs*. In Figure 4.12, the solver cannot provide concrete values of the uninterpreted type U for e_1 and op_1, it can only assign fresh, unconstrained variables (e.g., e and op). However, the triggering terms $\mathtt{some}(e)$ and $\mathtt{get}(op)$, which can be obtained from these fresh variables, are not sufficient to prove unsat; one would additionally need the rewriting $e_1 = \mathtt{get}(op_1)$, which cannot be identified

$$F_0\colon \quad \forall e_0 : \mathrm{U} :: \{\texttt{some}(e_0)\}\ \neg(\texttt{some}(e_0) = \texttt{none})$$
$$F_1\colon \quad \forall op_1 : \mathrm{U}, e_1 : \mathrm{U} :: \{\texttt{some}(e_1), \texttt{get}(op_1)\}\ \neg(\texttt{get}(op_1) = e_1) \vee (op_1 = \texttt{some}(e_1))$$
$$F_2\colon \quad \forall op_2 : \mathrm{U}, e_2 : \mathrm{U} :: \{\texttt{some}(e_2), \texttt{get}(op_2)\}\ \neg(op_2 = \texttt{some}(e_2)) \vee (\texttt{get}(op_2) = e_2)$$

FIGURE 4.12: Fragment of Gobra's [183] option types axiomatization. U is an uninterpreted type, `none` is a user-defined constant of type U. F_1 and F_2 have *multi-patterns* (discussed in Section 4.4.4). Synthesizing the triggering term $\texttt{dummy}(\texttt{some}(\texttt{get}(\texttt{none})))$ requires type-based constraints.

by our unification from Section 4.4.2. To address such scenarios, we extend the unification to also consider as *rhs* a constant or an uninterpreted function from the body of the similar formulas, which has the same *type* as the quantified variable from the left-hand side. For Figure 4.12, it will thus generate the rewritings $R = \{e_0 = \texttt{get}(op_2), e_1 = \texttt{get}(op_2), op_1 = \texttt{none}, op_2 = \texttt{none}\}$ (this is one of the alternatives). These type-based constraints allow us to synthesize the triggering term $T = \texttt{dummy}(\texttt{some}(\texttt{get}(\texttt{none})))$, which exposes the unsoundness from Gobra's [183] option types axiomatization.

Unification for sub-terms. Figure 4.13 shows an example which cannot be solved by any extension discussed so far, since it requires semantic reasoning: by applying `f` on both sides of the equality, one can learn from F_1 that $\texttt{f}(\texttt{g}(2020)) = \texttt{f}(\texttt{g}(2021))$. From F_0 though, $\texttt{f}(\texttt{g}(2020)) = 2020$ and $\texttt{f}(\texttt{g}(2021)) = 2021$, which implies that $2020 = 2021$, i.e., `false`. Our extended algorithm synthesizes the required triggering term $T = \texttt{dummy}(\texttt{f}(\texttt{g}(2020)), \texttt{f}(\texttt{g}(2021)))$ by applying the unification also to *sub-terms*; due to our restrictive shapes of the rewritings, the sub-terms can only be applications of *uninterpreted* functions. In Figure 4.13, trying to unify $\texttt{f}(\texttt{g}(x_0))$ does not produce any rewritings, as F_1 does not contain $\texttt{f}(\texttt{g})$. We thus unify the sub-term $\texttt{g}(x_0)$ with $\texttt{g}(2020)$ and $\texttt{g}(2021)$ and obtain the rewritings $R = \{x_0 = 2020, x_0 = 2021\}$. Together with the extension for combining multiple candidate terms described above, these rewritings provide sufficient information for the unsat proof to succeed. This unification is syntactic, but produces the triggering terms that would be obtained if the solver would apply some uninterpreted function present in the input on a learned predicate (the solver performs semantic reasoning automatically, but without generating new triggering terms).

$$F_0\colon \quad \forall x_0 : \mathrm{Int} :: \{\texttt{f}(\texttt{g}(x_0))\}\ \texttt{f}(\texttt{g}(x_0)) = x_0$$
$$F_1\colon \quad \texttt{g}(2020) = \texttt{g}(2021)$$

FIGURE 4.13: Formulas that set contradictory constraints on the uninterpreted function `f`. Synthesizing the triggering term $\texttt{dummy}(\texttt{f}(\texttt{g}(2020)), \texttt{f}(\texttt{g}(2021)))$ requires unification for sub-terms.

Alternative triggering terms. Our algorithm returns the *first* candidate term that successfully validates (Algorithm 4.1, line 18). However, it might also be useful to synthesize *alternative* triggering terms for the same input, since they may correspond to different completeness or soundness issues. Our tool provides this option and can also return *all* the triggering terms found within the given timeout.

All these extensions (individually or together with other extensions) allow us to complete the refutation proofs for particular benchmarks. Section 4.5 evaluates the impact of a few configurations of our technique, which can be obtained by enabling some of the extensions or by setting certain values for some of the additional parameters. Automatically determining which is the most suited configuration for a particular input is left as future work.

4.4.4 *Additional examples*

In this section, we illustrate our algorithm on various examples (including those from Figure 4.1 and Figure 4.2, and an example with nested quantifiers). We also explain how the algorithm supports quantifier-free formulas, synonym functions as patterns, multi-patterns, and alternative patterns.

Nested quantifiers. Our algorithm handles inputs with nested quantifiers as described in Section 4.4.2. We illustrate this aspect on the formulas from Figure 4.14, which axiomatize operations over lists of integers. The axioms F_3 and F_4 set contradictory constraints on `indexOf` when the element is not contained in the list. According to Algorithm 4.2, one of the clusters generated for F_3 is $C = \{F_2, F_0\}$, with the rewritings $R = \{l_3 = l_2, el_3 = el_2, l_2 = l_0\}$. The algorithm then computes the instantiations for F_0 and F_2; as F_2 contains nested quantifiers, we remove both of them and obtain: $\mathsf{Inst}[F_2] = \{\neg\texttt{isEmpty}(l_2), \texttt{isEmpty}(l_2) \wedge \neg\texttt{has}(l_2, el_2)\}$, $\mathsf{Inst}[F_0] = \{\neg(l_0 = \texttt{EmptyList}), (l_0 = \texttt{EmptyList}) \wedge \texttt{isEmpty}(l_0)\}$. The model of the corresponding G formula and R allow us to synthesize the required triggering term $T = \texttt{dummy}(\texttt{isEmpty}(\texttt{EmptyList}), \texttt{has}(\texttt{EmptyList}, 0))$.

F_0: $\forall l_0 : \mathrm{L} :: \{\texttt{isEmpty}(l_0)\}\ \neg(l_0 = \texttt{EmptyList}) \vee \texttt{isEmpty}(l_0)$

F_1: $\forall l_1 : \mathrm{L} :: \{\texttt{isEmpty}(l_1)\}\ \texttt{isEmpty}(l_1) \vee \texttt{has}(l_1, \texttt{f}_1(l_1))$

F_2: $\forall l_2 : \mathrm{L} :: \{\texttt{isEmpty}(l_2)\}\ \neg\texttt{isEmpty}(l_2) \vee \forall el_2 : \mathrm{Int} :: \{\texttt{has}(l_2, el_2)\}\ \neg\texttt{has}(l_2, el_2)$

F_3: $\forall l_3 : \mathrm{L}, el_3 : \mathrm{Int} :: \{\texttt{has}(l_3, el_3)\}\ \texttt{has}(l_3, el_3) \vee (\texttt{indexOf}(l_3, el_3) = -1)$

F_4: $\forall l_4 : \mathrm{L}, el_4 : \mathrm{Int} :: \{\texttt{indexOf}(l_4, el_4)\}\ \texttt{indexOf}(l_4, el_4) \geq 0$

FIGURE 4.14: Formulas that set contradictory constraints on the uninterpreted function `indexOf`. L is an uninterpreted type, `EmptyList` is a user-defined constant of type L. $\texttt{f}_1$ is a Skolem function, which replaces a nested existential quantifier. F_2 contains nested universal quantifiers.

F_0: $\forall x_0 : \text{Int} :: \{\texttt{len}(\texttt{nxt}(x_0))\}\ \texttt{len}(x_0) > 0$

F_1: $\forall x_1 : \text{Int} :: \{\texttt{len}(\texttt{nxt}(x_1))\}\ (\texttt{nxt}(x_1) = x_1) \vee (\texttt{len}(x_1) = \texttt{len}(\texttt{nxt}(x_1)) + 1)$

F_2: $\forall x_2 : \text{Int} :: \{\texttt{len}(\texttt{nxt}(x_2))\}\ \neg(\texttt{nxt}(x_2) = x_2) \vee (\texttt{len}(x_2) = 1)$

F_3: $\texttt{len}(7) \leq 0$

FIGURE 4.15: Boogie example from Figure 4.1 encoded in our input format. F_0–F_2 represent the axiom, while the quantifier-free formula F_3 is the negation of the assertion (to discharge the proof obligation, the verifier considers the axioms and the negation of the verification condition).

Quantifier-free formulas. Our algorithm iterates only over quantified conjuncts but leverages the additional information provided by quantifier-free formulas and includes them in the clusters even if the unification cannot find a rewriting (Algorithm 4.2, line 9). Since quantifier-free conjuncts can be seen as already instantiated formulas, we only have to cover all their disjuncts (Algorithm 4.1, line 7).

Boogie example. Figure 4.15 shows the example from Figure 4.1 encoded in our format. The quantifier-free formula F_3 (the negation of the verification condition) is similar to F_0 (they share the symbol `len`) and unifies through the rewritings $R = \{x_0 = 7\}$. We obtain the required triggering term $\texttt{dummy}(\texttt{len}(\texttt{nxt}(7)))$ from the model of $G = \neg F_0' \wedge \mathsf{Inst}[F_3][0] \wedge \bigwedge R = (\texttt{len}(x_0) \leq 0) \wedge (\texttt{len}(7) \leq 0) \wedge (x_0 = 7)$.

Dafny example. Our algorithm can synthesize various triggering terms that expose the unsoundness from Figure 4.2, depending on the values of its parameters. We explain here one, for $\sigma = 0.1$. For $\texttt{depth} = 0$, the algorithm checks each formula F_0–F_4 in isolation. As they are all individually satisfiable, it continues with $\texttt{depth} = 1$. To avoid redundant explanations, we present only the iteration for F_3.

F_3 shares at least two uninterpreted symbols with each of the other formulas, so there are various alternative rewritings: $s_3 = \texttt{Empty}(t_1)$, $s_3 = s_4$, $s_3 = \texttt{Build}(s_4, i_4, v_4, l_4)$, etc. As we consider clusters-rewritings pairs in which each quantified variable has maximum one rewriting, one such pair is $(C = \{F_4\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, l_4)\})$. F_4 has two disjuncts, therefore its instantiations are: $\mathsf{Inst}[F_4] = \{\texttt{typ}(s_4) = \texttt{Type}(\texttt{typ}(v_4), \neg(\texttt{type}(s_4) = \texttt{Type}(\texttt{typ}(v_4)) \wedge \texttt{Len}(\texttt{Build}(s_4, i_4, v_4, l_4)) = l_4\}$. From these instantiations and the rewritings R, we derive two formulas: $G_0 = \neg F_3' \wedge \mathsf{Inst}[F_4][0] \wedge \bigwedge R$, with the model $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $l_4 = 1$ and $G_1 = \neg F_3' \wedge \mathsf{Inst}[F_4][1] \wedge \bigwedge R$, with the model $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $l_4 = -1$, where s, s', and v are fresh variables of type U. (We use indexes for the G formulas to refer to them later.) We then construct the candidate triggering terms from the patterns of the formulas F_3 and F_4. We replace s_3 by its *rhs* in the rewriting, i.e., $\texttt{Build}(s_4, i_4, v_4, l_4)$, and all the other quantified variables by their constants from the model. The result after removing redundant terms is: $T_0 = \texttt{dummy}(\texttt{Len}(\texttt{Build}(t, 0, v, 1)))$ and $T_1 = \texttt{dummy}(\texttt{Len}(\texttt{Build}(t, 0, v, -1)))$. Since the validation step fails for both T_0 and T_1, we continue with the other (C, R) pairs, the remaining quantified conjuncts, and their similarity clusters.

$$F: \quad \forall a: \text{Int}, b: \text{Int}, size: \text{Int} :: \{\texttt{both\_ptr}(a, b, size)\}$$
$$\texttt{both\_ptr}(a, b, size) * size \leq a - b$$

FIGURE 4.16: Inconsistent formula from a VCC/HAVOC [140, 45] benchmark from SMT-COMP [168], which cannot be proved unsat by MBQI. Our synthesized triggering term $\texttt{dummy}(\texttt{both\_ptr}(-2, -1, 0))$ allows E-matching to refute it.

If no candidate term is sufficient to prove unsat, our algorithm expends the clusters. To scale to real-world axiomatizations, it efficiently reuses the results from the previous iterations; i.e., it prunes the search space if a previously synthesized formula G is unsatisfiable and it strengthens G if it is satisfiable. The pair $(C = \{F_4\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, l_4)\})$ can be extended to $(C = \{F_4, F_1\}, R = \{s_3 = \texttt{Build}(s_4, i_4, v_4, l_4), s_4 = \texttt{Empty}(t_1), t_1 = \texttt{typ}(v_4)\})$, as F_1 is similar to F_4 through the rewritings $R = \{s_4 = \texttt{Empty}(t_1), t_1 = \texttt{typ}(v_4)\}$. We thus conjoin the instantiation of F_1 and the two additional rewritings to the formulas G_0 and G_1 from the previous iteration. This is equivalent to recomputing the similarity cluster, the rewritings, and the combinations of instantiations. We then obtain: $G_0' = G_0 \wedge (\texttt{type}(\texttt{Empty}(t_1)) = \texttt{Type}(t_1)) \wedge (s_4 = \texttt{Empty}(t_1)) \wedge (t_1 = \texttt{typ}(v_4))$, which is unsatisfiable, and $G_1' = G_1 \wedge (\texttt{type}(\texttt{Empty}(t_1)) = \texttt{Type}(t_1)) \wedge (s_4 = \texttt{Empty}(t_1)) \wedge (t_1 = \texttt{typ}(v_4))$ with the model: $s_3 = s$, $s_4 = s'$, $i_4 = 0$, $v_4 = v$, $l_4 = -1$, $t_1 = t$, where s, s', v, and t are fresh variables of types U and V. From this model and the rewritings we construct the triggering term $T = \texttt{dummy}(\texttt{Len}(\texttt{Build}(\texttt{Empty}(\texttt{typ}(v)), 0, v, -1)))$, which is sufficient to expose the inconsistency between F_3 and F_4.

VCC/HAVOC example. Figure 4.16 presents a fragment of a benchmark which could be solved by our algorithm, but could not be proved unsat by MBQI (which is a quantifier-instantiation algorithm that *does not* rely on patterns; Section 4.5 provides additional experimental results and a detailed comparison with alternative refutation techniques). F, which was part of a set of 160 formulas, is inconsistent by itself: when $size = 0$, E-matching can refute it for any integer values a, b, such that $a \leq b$. Our algorithm synthesizes the required triggering term in $\approx$7 s because it initially considers each quantified conjunct in isolation. The formula $G = \neg F' = \texttt{both\_ptr}(a, b, size) * size > a - b$ is satisfiable and the simplest models the solver can provide (without assigning an interpretation to the uninterpreted function `both_ptr`) all include $size = 0$.

Synonym functions as patterns. For the examples discussed so far, the functions used as patterns were also present in the body of the quantifiers. However, to have better control over the instantiations, one can also write formulas where the patterns are additional uninterpreted functions, which do not appear in the bodies. Such patterns are not uncommon in proof obligations. Figure 4.17 shows an example, which uses the synonym functions technique [107] to avoid matching loops. `sum` and `sum_syn` compute the sum of the elements of a sequence, between a lower and an upper bound. The two functions are identical (according to F_0), but only

$$
\begin{aligned}
F_0\colon&\quad \forall s_0\colon \mathrm{ISeq}, l_0\colon \mathrm{Int}, h_0\colon \mathrm{Int} :: \{\mathtt{sum}(s_0,l_0,h_0)\}\ \mathtt{sum}(s_0,l_0,h_0) = \mathtt{sum\_syn}(s_0,l_0,h_0)\\
F_1\colon&\quad \forall s_1\colon \mathrm{ISeq}, l_1\colon \mathrm{Int}, h_1\colon \mathrm{Int} :: \{\mathtt{sum}(s_1,l_1,h_1)\}\ \neg(l_1 \geq h_1) \vee \mathtt{sum\_syn}(s_1,l_1,h_1) = 0\\
F_2\colon&\quad \forall s_2\colon \mathrm{ISeq}, l_2\colon \mathrm{Int}, h_2\colon \mathrm{Int} :: \{\mathtt{sum}(s_2,l_2,h_2)\}\ \neg(l_2 \leq h_2) \vee\\
&\qquad (\mathtt{sum\_syn}(s_2,l_2,h_2) = \mathtt{sum\_syn}(s_2,l_2+1,h_2) + \mathtt{seq.nth}(s_2,l_2))\\
F_3\colon&\quad \mathtt{seq.nth}(\mathtt{empty},0) = -1
\end{aligned}
$$

FIGURE 4.17: Formulas with synonym functions as patterns that axiomatize sequence comprehensions and set contradictory constraints on the uninterpreted function `sum_syn`. ISeq is a user-defined type, `empty` is a user-defined constant of type ISeq (i.e., the empty sequence).

`sum` is used as a pattern. For equal bounds, F_1 and F_2 set contradictory constraints on the interpretation of `sum_syn`. `seq.nth` returns the n-th element of the sequence. Using the information from the quantifier-free formula F_3, our algorithm generates the triggering term $T = \mathtt{dummy}(\mathtt{sum}(\mathtt{empty},0,0), \mathtt{sum}(\mathtt{empty},0+1,0))$. The term "$0+1$" comes from the rewriting $l_0 = l_2 + 1$. The addition is a built-in function but is used as an argument to the uninterpreted function `sum_syn`, thus, it is supported by our unification. Our algorithm is syntactic, so it does not perform arithmetic operations, it just substitutes l_2 with its value from the model. The solver then performs theory reasoning and concludes unsat.

Multi-patterns and alternative patterns. SMT solvers allow patterns to contain multiple terms, all of which must be present to perform an instantiation. F_1 in Figure 4.18 has such a *multi-pattern* and can be instantiated only when triggering terms that match both $\{\mathtt{g}(b_1)\}$ *and* $\{\mathtt{f}(x_1)\}$ are present in the SMT run. Our algorithm directly supports multi-patterns, as the procedure candidateTerm instantiates *all* the patterns from the given cluster (see Algorithm 4.3, line 9). For the example from Figure 4.18, our technique synthesizes the triggering term $T = \mathtt{dummy}(\mathtt{f}(7), \mathtt{g}(b))$ from the rewritings $R = \{x_0 = x_1\}$ and the model of the formula $G = \neg F_0' \wedge \mathsf{Inst}[F_1][1] \wedge \bigwedge R = (\mathtt{f}(x_0) = 7) \wedge (\neg\mathtt{g}(b_1) \wedge \mathtt{f}(x_1) = x_1) \wedge (x_0 = x_1)$. Here b is a fresh, unconstrained variable of the uninterpreted type B.

Formulas can also contain *alternative patterns*. For example, the quantified formula $\forall x\colon \mathrm{Int} :: \{\mathtt{f}(x)\}\ \{\mathtt{h}(x)\}\ \mathtt{f}(x) \neq 7 \vee \mathtt{h}(x) = 6$ is instantiated only if there exists a triggering term that matches $\{\mathtt{f}(x)\}$ *or* one that matches $\{\mathtt{h}(x)\}$. Our algorithm does not differentiate between multi-patterns and alternative patterns, thus

$$
\begin{aligned}
F_0\colon&\quad \forall x_0\colon \mathrm{Int} :: \{\mathtt{f}(x_0)\}\ \mathtt{f}(x_0) \neq 7\\
F_1\colon&\quad \forall b_1\colon \mathrm{B}, x_1\colon \mathrm{Int} :: \{\mathtt{g}(b_1), \mathtt{f}(x_1)\}\ \mathtt{g}(b_1) \vee (\mathtt{f}(x_1) = x_1)\\
F_2\colon&\quad \forall b_2\colon \mathrm{B} :: \{\mathtt{g}(b_2)\}\ \neg\mathtt{g}(b_2)
\end{aligned}
$$

FIGURE 4.18: Formulas that set contradictory constraints on the uninterpreted function `f`. B is an uninterpreted type. F_1 has a multi-pattern.

it always synthesizes the arguments for *all* the patterns of a cluster. For alternative patterns, this results in an over-approximation of the set of necessary triggering terms. However, the minimization step (performed before returning the triggering term that successfully validates), removes the unnecessary terms.

4.5 EVALUATION

Evaluating our work requires benchmarks with known triggering issues (i.e., for which E-matching yields *unknown*). Since there is no publicly available suite, in Section 4.5.1 we used manually-collected benchmarks from four verifiers [106, 124, 162, 183]. Our algorithm succeeded for 65.6%. To evaluate its applicability to other verifiers, in Section 4.5.2 we used SMT-COMP [168] inputs. As they were not designed to expose triggering issues, we developed a filtering step to automatically identify the subset that falls into this category. The results show that our algorithm is suited also for [15, 45, 140]. Section 4.5.3 illustrates that our triggering terms are simpler than the unsat proofs produced by quantifier instantiation and refutation techniques, enabling one to fix the root cause of the revealed issues.

Setup. We used Z3 (4.8.10) [187] to infer the patterns, generate the models, and validate the candidate terms. However, our tool can be used with any solver that supports E-matching and exposes the inferred patterns. We used Z3's `NNF` tactic to transform the inputs into NNF and locality-sensitive hashing to compute the clusters. We fixed Z3's random seeds to arbitrary values (`sat.random_seed` to 488, `smt.random_seed` to 599, and `nlsat.seed` to 611). We set the (soft) timeout to 600 s and the memory limit to 6 GB per run and used a 1 s timeout for obtaining a model and for validating a candidate term. The experiments were conducted on a Linux server with 252 GB of RAM and 32 Intel Xeon CPUs at 3.3 GHz.

4.5.1 *Effectiveness on benchmarks with triggering issues*

First, we used *manually*-collected benchmarks with known triggering issues from 4 state-of-the-art program verifiers: Dafny [106], F* [162], Gobra [183], Viper [124]. We reconstructed 4, respectively 2 inconsistent axiomatizations from Dafny and F*, based on the changes from the repositories and the messages from the issue trackers; we obtained 11 inconsistent axiomatizations of arrays and option types from Gobra's developers and collected 15 incompleteness issues from Viper's test suite [176], with at least one assertion needed only for triggering (we removed these assertions from the benchmarks, as our work is expected to find the triggering terms automatically). The Viper files contain algorithms for arrays, binomial heaps, binary search trees, and regression tests. The input sizes (minimum-maximum number of formulas or quantifiers) are shown in Table 4.1, columns 3–4.

Configurations. We ran our tool with five configurations, to also analyze the impact of its parameters (see Algorithm 4.1 and Section 4.4.3). The default configuration Co has: $\sigma = 0.3$ (similarity threshold), $\beta = 64$ (batch size, i.e., the number

Source	#	#F min-max	#∀ min-max	C0 default	C1 σ=0.1	C2 β=1	C3 type	C4 σ=0.1 ∧ sub	Our work	Z3 MBQI	CVC4 enum inst	Vampire CASC ∧ Z3
Dafny	4	6 - 16	5 - 16	1	1	1	1	0	**1**	1	0	2
F*	2	18 - 2 388	15 - 2 543	1	1	1	1	2	**2**	1	0	2
Gobra	11	64 - 78	50 - 63	5	10	1	7	10	**11**	6	0	11
Viper	15	84 - 143	68 - 203	7	5	3	5	5	**7**	11	0	15
Total	**32**								**21 (65.6%)**	**19 (59.3%)**	**0 (0%)**	**30 (93.7%)**

σ = similarity threshold; β = batch size; **type** = type-based constraints; **sub** = sub-terms **C0**: $\sigma = 0.3$; $\beta = 64$; ¬**type**; ¬**sub**

TABLE 4.1: Results on verification benchmarks with known triggering issues. The columns from left to right show: the source of the benchmarks, the number of files (#), their number of conjuncts (#F) and of quantifiers (#∀), the number of files for which five different configurations of our algorithm (C0 – C4) synthesized suited triggering terms, our results across all configurations, the number of unsat proofs generated by Z3 (with MBQI [78]), CVC4 (with enumerative instantiation [136]), and Vampire [98] (in CASC mode [161], using Z3 for ground theory reasoning). The columns marked with grey represent E-matching-based algorithms; only those can be soundly used by verifiers whose SMT encodings have patterns, i.e., are designed for E-matching.

of candidate terms validated together), ¬type (no type-based constraints), ¬sub (no unification for sub-terms). The other configurations differ from C0 in the parameters shown in Table 4.1. All configurations use $\delta = 2$ (maximum transitivity depth), $\mu = 4$ (maximum number of different models), and 600 s timeout per file.

Results. Columns 5–9 in Table 4.1 show the number of files solved by each configuration, column 10 summarizes those solved by at least one. Overall, we found suited triggering terms for 65.6%, including all F* and Gobra benchmarks. An F* unsoundness exposed by all configurations in ≈60 s is given in Figure 4.9. It required two developers to be manually diagnosed based on a bug report [72]. A simplified Gobra axiomatization for option types is shown in Figure 4.12; the entire axiomatization (considered in Table 4.1) was solved only by C4 in ≈13 s. Gobra's team spent one week to identify some of the issues. As our triggering terms for F* and Gobra were similar to the manually-written ones, we believe they could have reduced the human effort in localizing and fixing the errors.

Our algorithm synthesized missing triggering terms for 7 Viper files, including the array maximum example [5], for which E-matching could not prove that the maximal element in a strictly increasing array of size 3 is its last element. Our triggering term `loc(a,2)` (`loc` maps arrays and integers to heap locations) can be added by a *user* of the verifier to their postcondition. A *developer* can fix the root cause of the incompleteness by including a generalization of the triggering term to arbitrary array sizes: `len(a) != 0 ==> x == loc(a, len(a)-1).val` (`val` allows one to access the value at the corresponding heap location). Both fixes result in E-matching refuting the proof obligation in under 0.1 s. We also exposed another case where Boogie (which is used by Viper) is sound only modulo patterns (as in Figure 4.3), i.e., the unsoundness is visible only at the SMT level.

Source	#	#F min-max	#$\forall$ min-max	C0 default	C1 σ=0.1	C2 β=1	C3 type	C4 σ=0.1 $\wedge$ sub	Our work	Z3 MBQI	CVC4 enum inst	Vampire CASC $\wedge$ Z3
Spec#	33	28 - 2 363	25 - 645	16	16	14	16	15	**16**	16	0	29
VCC/Havoc	14	129 - 1 126	100 - 1 027	11	9	5	11	9	**11**	12	0	14
Simplify	1	256	129	0	0	0	0	0	**0**	1	0	0
BWI	13	189 - 384	198 - 456	1	1	2	1	1	**2**	12	0	12
Total	**61**								**29 (47.5%)**	**41 (67.2%)**	**0 (0%)**	**55 (90.1%)**

σ = similarity threshold; β = batch size; **type** = type-based constraints; **sub** = sub-terms — **C0**: $\sigma = 0.3$; $\beta = 64$; $\neg$**type**; $\neg$**sub**

TABLE 4.2: Results on SMT-COMP inputs. The columns have the structure from Table 4.1.

As Table 4.1 shows, configurations with smaller σ (C1 and C4) were particularly important for some of the F* and Gobra benchmarks. Our algorithm starts with the given σ and if it does not find the required triggering terms, it decreases σ by 0.1 and reiterates. Thus C0 also covers the case $\sigma = 0.1$, if the overall timeout is large enough. However, always starting with a small σ may prevent our algorithm from synthesizing the triggering terms, since the number of rewritings it has to explore is considerably high. The extensions for unifying sub-terms (C4) and identifying type-based constraints (C3) were also needed for one, respectively two input files.

4.5.2 *Effectiveness on SMT-COMP benchmarks*

Next, we considered 61 SMT-COMP [168] benchmarks from Spec# [15], VCC [140], Havoc [45], Simplify [63], and the Bit-Width-Independent (BWI) encoding [127]. These were selected *automatically* and are summarized in Table 4.2. To obtain them, we proceeded as described below.

Benchmarks selection. We collected 27 716 benchmarks from SMT-COMP 2020 (single query track) [168], with ground truth *unsat* and at least one pattern (as this suggests they were designed for E-matching). We then ran Z3 to infer the missing patterns and to transform the formulas into NNF and removed all benchmarks for which the inference or the transformation did not succeed within 600 s per file and 4 s per formula. We also removed the files with features not yet supported by PySMT [77], the parsing library used in our experiments (e.g., sort signatures in datatypes declarations), but we did extend PySMT to handle, e.g., patterns and overloaded functions. This filtering resulted in 6 481 benchmarks. We then ran E-matching and kept only those 61 examples that could not be solved within 600 s due to incompleteness in instantiating quantifiers (our work only targets this incompleteness, but the SMT-COMP suite also contains other solving challenges).

Results. The results are shown in Table 4.2, which follows the structure of Table 4.1. Our algorithm enabled E-matching to refute 47.5% of the files, most of them from Spec# and VCC/Havoc. We manually inspected some BWI benchmarks (for which the algorithm had worse results) and observed that the validation step times out even with a much higher timeout. This shows that some candidate terms trigger matching loops and explains why C2 (which validates them individually) solved one more file. Extending our algorithm to avoid matching loops, by construction,

is left as future work. The other configurations did not prove to be better than Co for these SMT-COMP inputs.

4.5.3 *Comparison with unsatisfiability proofs*

As an alternative to our work, tool developers could try to *manually* identify triggering issues from refutation proofs, but unless these are generated by E-matching-based algorithms, they *do not* consider patterns and require expert knowledge to be understood. (Section 4.8 discusses the challenges of automating this process.)

Columns 11–13 in Table 4.1 and Table 4.2 show the number of unsatisfiability proofs produced by Z3 with MBQI [78], CVC4 [17] with enumerative instantiation [136] (an algorithm based on E-matching, used when E-matching saturates), and the first-order theorem prover Vampire [98], using Z3 for ground theory reasoning [134] and the CASC [161] portfolio mode with competition presets. CVC4 failed for all examples (it cannot construct proofs for quantified logics), Vampire refuted most of them. Our algorithm outperformed MBQI for F* and Gobra and had similar results for Dafny, Spec#, and VCC/Havoc. All five configurations solved two VCC/Havoc files not solved by MBQI (Figure 4.16 presents one).

In terms of complexity, our triggering terms are much simpler and directly highlight the root cause of the issues. For Viper's array maximum example from Section 4.5.1, our triggering term `loc(a,2)` is easier to understand than MBQI's proof (which has 2 135 lines and over 700 reasoning steps) and then Vampire's proof (with 348 lines and 101 inference steps). Other proofs have similar sizes.

As Vampire and MBQI *do not* consider patterns, they cannot replace our technique. Most deductive verifiers [3, 7, 15, 45, 66, 106, 163] employ E-matching for discharging their proof obligations, since E-matching is the most efficient SMT algorithm for program verification [78] (the vast majority of the SMT-COMP benchmarks we initially collected were also directly refuted by E-matching). It is thus important to help the developers use the algorithm of their choice and return sound results even if they rely on patterns for soundness (as in Figure 4.3). Therefore, the only approach from Table 4.1 and Table 4.2 comparable with ours is enumerative instantiation; the others solve a *different* problem.

As our algorithm accepts as input an SMT formula, it can also produce triggering terms required only at the SMT level, but which cannot be encoded into the input language of the verifier (e.g., Boogie), since they are rejected by the type system. However, such triggering terms can be filtered out, as lifting them to the input language is mostly straightforward (we performed this step manually in our experiments, to identify the cases of soundness modulo patterns; automating this process is a possible future extension). However, this is not the case for refutation proofs, whose back translation to the source language is an open research problem.

To enable the developers debug the axiomatizations or fix the incompleteness more efficiently, our tool can also generate *multiple* triggering terms (as explained

in Section 4.4.3). It can thus reveal *multiple* triggering issues for the same input formula, information which cannot be directly obtained from unsatisfiability proofs.

4.5.4 *Threats to validity*

We identified the following two threats to the validity of our experiments:

Non-determinism. The SMT solvers use randomized algorithms, which can cause non-determinism. To mitigate this problem, we fixed all the available random seeds and used the same seeds in all the phases of our evaluation (i.e., for inferring the patterns, pre-filtering via E-matching, running our tool and MBQI).

Benchmarks selection. We relied on Z3's E-matching algorithm to select examples with incompleteness in instantiating quantifiers. An implementation of E-matching from another solver could have led to different files. To avoid biases, we used Z3 in all the experiments.

4.6 OPTIMIZATIONS

In this section, we present various optimizations implemented in our tool, which allow the algorithm to scale to real-world verification benchmarks.

Grammar. The grammar from Figure 4.6 allows us to simplify the presentation of the algorithm. However, eliminating conjunctions by applying distributivity and splitting (as described in Section 4.4.1) can result in an exponential increase in the number of terms and introduce redundancy, affecting the solver's performance. Conjunction elimination is not implemented in Z3's `NNF` tactic (used in our evaluation, see Section 4.5), thus it is not performed automatically. We apply this transformation only at the top level, i.e., we do not recursively distribute disjunctions over conjunctions. For this reason, the input conjuncts F supported by our tool can actually contain conjunctions, in which case we use an extended algorithm when computing the instantiations, to ensure that all the resulting G formulas are still quantifier-free. The number of conjuncts and the number of quantifiers reported in Table 4.1 and Table 4.2 were computed *before* applying distributivity, thus they are not artificially increased.

Rewritings. The restrictive shapes of our rewritings (from Section 4.4.2, step 2), ensure that their number is finite, because if it exists, the most general unifier is unique up to variable renaming, i.e., substitutions of the type $\{x_i \to x_j, x_j \to x_i\}$ [10]. (Such substitutions are rewritings of the shapes (1) and (2) where *rhs* is also a quantified variable.) However, for most practical examples, the number of rewritings is very large, thus our implementation identifies them *lazily*, in increasing order of cardinality. If a rewriting $r \in R$ leads to an unsat G formula for some instantiations, then we discard all the subsequent G formulas that contain r and the same instantiations (they will also be unsatisfiable). To make sure that the al-

gorithm terminates within a given amount of time, in our experiments we bound the number of G formulas derived for each quantified conjunct F to 100.

Instantiations. Our implementation computes lazily the Cartesian product of the instantiations (Algorithm 4.1, line 9) since it can also have a high number of elements. However, many of them are in practice unsatisfiable; our tool efficiently identifies trivial conflicts (e.g., $\neg D_i \wedge D_i$), pruning the search space accordingly.

Candidate terms. To improve the performance of our algorithm, we keep track of all the candidate triggering terms that failed to validate (i.e., of the models from which they were synthesized). Then, we add constraints (similar to the conjunct $\neg$`model` from Algorithm 4.1, line 19) to ensure the solver does not provide previously-seen models for the quantified variables from the same set of patterns.

4.7 LIMITATIONS

In the following, we discuss the limitations of our approach and possible solutions.

Applicability. Our algorithm effectively addresses a common cause of failed unsatisfiability proofs in program verification, i.e., missing triggering terms. Other causes (e.g., incompleteness in the solver's decision procedures due to undecidable theories) are beyond the scope of our work. Also, our algorithm is tailored to *unsatisfiability* proofs; satisfiability proofs cannot be reduced to unsatisfiability proofs by negating the input, because the negation cannot usually be encoded in SMT (as we have illustrated in Section 4.3).

SMT solvers. Our algorithm synthesizes triggering terms as long as the SMT solver can find models for our quantifier-free formulas. However, solvers are incomplete, i.e., they can return *unknown* and generate only *partial models*, which are not guaranteed to be correct. Nonetheless, we also use partial models, as the validation step (step 4 in Figure 4.4) ensures that they do not lead to false positives.

Patterns. Since our algorithm is based on patterns (provided or inferred), it will not succeed if they do not permit the necessary instantiations. For example, the formula $\forall x\colon \mathrm{Int}, y\colon \mathrm{Int} :: x = y$ is unsatisfiable. However, the SMT solver cannot automatically infer a pattern from the body of the quantifier, since equality is an interpreted function and must not occur in a pattern. Thus E-matching (and implicitly our algorithm) cannot solve this example, unless the user provides as pattern some uninterpreted function that mentions both x and y (e.g., $\mathtt{f}(x, y)$).

Bounds and rewritings. Synthesizing triggering terms is generally undecidable. We ensure termination by bounding the search space through various customizable parameters, thus our algorithm misses results not found within these bounds. We also only unify applications of uninterpreted functions, which are common in verification. Efficiently supporting interpreted functions (especially equality) is very challenging for inputs with a small number of types (e.g., from Boogie [14]).

Despite these limitations, our algorithm effectively identifies the triggering terms required in practical examples, as we have experimentally shown in Section 4.5.

4.8 CONSTRUCTING TRIGGERING TERMS FROM UNSATISFIABILITY PROOFS

In this section, we describe an alternative approach for synthesizing triggering terms for E-matching, starting from the unsatisfiability proofs generated by other SMT algorithms or refutation techniques. We explored this idea together with Strebel in his Bachelor thesis [157] for MBQI [78] and Vampire [98], as a follow up of the work presented so far.

Explaining unsatisfiability proofs through examples. Since the proofs of MBQI and Vampire are complex (see Section 4.5.3) and do not follow a common format, we designed a technique to explain them to (non-expert) users through simple *examples* [157]. For an input `I` (defined based on the grammar from Figure 4.6), an example consists of a set of *quantified* conjuncts F of `I` *instantiated* for the concrete values used in the proofs, together with the quantifier-free conjuncts relevant for exposing the contradiction. An example is thus a quantifier-free, unsatisfiable formula, which highlights *one* scenario in which the solver cannot find an interpretation for (some of) the uninterpreted functions from the original input `I`.

Figure 4.19 shows the example constructed according to [157] from the proof produced by MBQI for the inconsistent axiomatization from Figure 4.5. $F_{0\,Inst}$ and $F_{1\,Inst}$ are the instatiations of F_0 and F_1 for the fresh variables x_0 and x_1; E_0 and E_1 set their values to $\mathtt{g}(7)$ and 7, respectively, terms present in the proof.

Synthesizing triggering terms from examples. The previously-constructed examples can be also used for synthesizing triggering terms [157], by replacing the quantified variables from the patterns of those conjuncts F which were instantiated in the example (F_0 and F_1 from Figure 4.5) with their values from the proofs (i.e., the right-hand side of the equalities E_0 and E_1 in Figure 4.19). In this way, we obtain the triggering term $\mathtt{dummy}(\mathtt{f}(\mathtt{g}(7)), \mathtt{g}(7))$, which enables E-matching to complete the proof when added to Figure 4.5. This term can be further minimized.

$$
\begin{aligned}
E_0 &: \quad x_0 = \mathtt{g}(7) \\
E_1 &: \quad x_1 = 7 \\
F_{0\,Inst} &: \quad \mathtt{f}(\mathtt{g}(x_1)) = x_1 \\
F_{1\,Inst} &: \quad \mathtt{f}(x_0) \neq 7
\end{aligned}
$$

FIGURE 4.19: Example (constructed according to [157]) showing a case in which the formulas from Figure 4.5 set contradictory constraints on the uninterpreted function `f`. x_0 and x_1 are fresh integer variables and represent concrete instantiations of the quantified variables with the same names from Figure 4.5.

Source	#	Z3 MBQI	Z3 API MBQI	Constructed examples	Triggering terms from examples
Dafny	4	1	1	1	**1**
F*	2	1	1	0	**0**
Gobra	11	6	6	6	**6**
Viper	15	11	7	7	**7**
Total	**32**	**19 (59.3%)**	**15 (46.87%)**	**14 (43.75%)**	**14 (43.75%)**

TABLE 4.3: Results of the algorithm for synthesizing triggering terms from MBQI's unsatisfiability proofs on the benchmarks from Table 4.1. The last four columns show: the number of unsat proofs generated by Z3 with MBQI (used from command line, as in Table 4.1), the number of unsat proofs generated by Z3 API with MBQI, the number of examples constructed according to our approach from [157] from the Z3 API's proofs, and the number of cases in which we synthesized triggering terms from the previously-constructed examples.

Practical challenges. While the approach described above provides a theoretical solution for constructing triggering terms from unsatisfiability proofs, there are various practical challenges one needs to overcome. First, MBQI and Vampire use different refutation algorithms, which are reflected in their proof format; thus the actual construction of the examples is prover-specific. Second, while MBQI's proofs contain *explicit* quantifier instantiations, the superposition and resolution calculi employed by Vampire lead only to *implicit* instantiations. We defined various *syntactic heuristics* [157] to convert them into explicit instantiations, which proved to be useful for simple, manually-written formulas. However, when evaluated on more complex benchmarks (from Table 4.1 and Table 4.2), they enabled the construction of the examples only for 6 out of 61 SMT-COMP files; Vampire's proofs for Dafny, F*, Gobra, and Viper (i.e., all the benchmarks from Table 4.1) could not be handled. The inputs currently not supported by our technique [157] would have required a combination of heuristics, and richer, semantic information from the proofs (see Section 6 from [157] for a detailed discussion of the results).

We extended the implementation from [157] to synthesize triggering terms from the examples obtained from MBQI's proofs and evaluated it on the files from Table 4.1. The results are summarized in Table 4.3. In our previous experiments (from Section 4.5.1 and Section 4.5.2), we ran Z3 with MBQI from command line, while in [157] we used the Z3 API to generate and parse the proofs (to avoid implementing our own parser). We then observed that the Z3 API is less performant than the command line [157], which our results from Table 4.3 can also confirm (see columns 3 and 4). We could construct examples for all but one file (column 5) within a 600 s timeout. The unsupported F* benchmark has 2 388 formulas and our tool [157] is not optimized to efficiently extract information from very large input files. Nevertheless, we could synthesize triggering terms that enable E-matching to complete the proofs from all the constructed examples (column 6). For this, we had to disable the minimization of the examples, which is performed based on the unsat core generated by MBQI [157], since a minimal MBQI example may not con-

tain instantiations for some quantified variables that are relevant for E-matching. Recall from Section 4.4.1 that a valid pattern should mention *all* the quantified variables, thus a triggering term has to provide concrete values for all of them.

Overall, we believe our technique from [157] is an important first step in automating the time-consuming process of understanding the unsatisfiability proofs of MBQI and Vampire, but requires additional effort to be useful for E-matching.

4.9 RELATED WORK

To the best of our knowledge, no other approach automatically produces the information needed by users or developers of program verifiers to remedy the effects of overly restrictive patterns. Quantifier instantiation and refutation techniques (discussed next) can produce unsatisfiability proofs, but these are much more complex than our triggering terms (as we have shown in Section 4.5.3).

Quantifier instantiation techniques. *Model-based quantifier instantiation* (MBQI) [78] was designed for sat formulas. It checks if the models obtained for the quantifier-free part of the input satisfy the quantifiers, whereas we check if the synthesized triggering terms obtained for some interpretation of the uninterpreted functions generalize to all interpretations. In some cases, MBQI can also generate unsatisfiability proofs, but they require expert knowledge to be understood; our triggering terms are much simpler. *Counterexample-guided quantifier instantiation* [137] is a technique for sat formulas, which synthesizes computable functions from logical specifications. It is applicable to functions whose specifications have explicit syntactic restrictions on the space of possible solutions, which is usually not the case for axiomatizations. Thus the technique cannot directly solve the complementary problem of proving the soundness of the axiomatization.

E-matching-based approaches. Rümmer [139] proposed a *calculus* for first-order logic modulo linear integer arithmetic that integrates constraint-based free variable reasoning with E-matching. Our algorithm does not require reasoning steps, so it is applicable to formulas from all the logics supported by the SMT solver. *Enumerative instantiation* [136] is an approach that exhaustively enumerates ground terms from a set of ordered, quantifier-free terms from the input. It can be used to refute formulas with quantifiers, but not to construct proofs (see Section 4.5.3). Our algorithm derives quantifier-free formulas and synthesizes the triggering terms from their models, even if the input does not have a quantifier-free part; we use also syntactic information (from the rewritings) to construct complex triggering terms.

Theorem provers. First-order theorem provers (e.g., Vampire [98]) also generate refutation proofs. More recent works combine a superposition calculus with theory reasoning [134, 179], integrating SAT/SMT solvers with theorem provers. We also use unification, but to synthesize triggering terms required by E-matching. However, our triggering terms are much simpler than Vampire's proofs and can be used to improve the triggering strategies for all future runs of the verifier.

Testing axiomatizations. Ahn and Denney [1] proposed a technique that identifies inconsistencies in quantified axiomatizations *without* patterns. The work also requires a computational model of the axioms, which includes interpretations for all the function symbols. Thus, it cannot be applied to axiomatizations with uninterpreted functions and types, which are very common in program verification. The technique tests each axiom in isolation, so it cannot find non-trivial inconsistencies caused by the interaction between axioms. Our approach is fully automatic and detects complex errors by identifying sharing constraints between formulas and synthesizing triggering terms from clusters of similar formulas.

Detecting matching loops. Becker et al. [19] dynamically detect performance issues in quantified SMT formulas that already include triggering terms, by identifying too *permissive* patterns that lead to matching loops. Our work targets soundness and completeness errors and synthesizes the triggering terms required to refute SMT inputs with overly *restrictive* patterns.

4.10 CONCLUSIONS

In this chapter, we presented the first automated technique that enables developers and users of verifiers remedy the effects of overly restrictive patterns. As discharging proof obligations and identifying inconsistencies in axiomatizations require the SMT solver to prove the unsatisfiability of a formula via E-matching, we developed a novel algorithm for synthesizing triggering terms that allow the solver to complete the proof. Our approach is effective for a diverse set of verifiers, and can significantly reduce the human effort in localizing and fixing triggering issues.

5

CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize the main contributions of our work (Section 5.1) and discuss research directions we would like to explore in the future (Section 5.2).

5.1 CONCLUSIONS

In this dissertation, we have presented various techniques for automatically identifying soundness and completeness/precision errors in different types of program analysis tools. Chapter 2 focuses on numerical abstract domains, the main components of state-of-the-art static analyses. Chapter 3 describes approaches for testing SMT and automata-based solvers. Chapter 4 shows that for SMT-based program verifiers, the incompleteness and the undetected unsoundness in their axiomatizations can be solved by enabling E-matching to complete unsatisfiability proofs. Next, we revisit the challenges from Section 1.2 and summarize our solutions.

Input data. Both Chapter 2 and Chapter 3 describe techniques for automatically constructing the input data. To identify errors in static analyzers in a unit testing manner, our input data consists of abstract domain elements. We generate them in two steps: first, we create a pool of representative elements with maximum one constraint, then we obtain more complex ones by applying a sequence of domain operations. This approach produces valid and diverse abstract elements. We use a similar idea in Chapter 3, where we start with simple SMT formulas that are satisfiable or unsatisfiable by construction and derive more complex ones through satisfiability-preserving transformations. Chapter 4 assumes that the input data is given and consists of the proof obligations or the axiomatizations of a verifier.

Test oracles. In Chapter 2, the test oracles are represented by the 46 algebraic properties we have derived from the abstract interpretation literature. Chapter 3 uses as test oracles the known truth value of the SMT formulas (which is preserved by our transformations), as well as the minimal unsat core for unsatisfiable formulas (which is unique by construction). Chapter 4 considers the E-matching implementation from Z3 as the test oracle, i.e., it uses it to determine which of the inputs cannot be refuted due to incompleteness in instantiating quantifiers.

Easy error reporting. The input construction techniques from Chapter 2 and Chapter 3 facilitate error reporting, as the issues are usually exposed by small test cases. However, the same bug can be revealed by multiple inputs; to avoid reporting the same problem multiple times, we manually identified similar test cases. We believe this process can be automated, as we explain in Section 5.2. Our algorithm from Chapter 4 also minimizes the synthesized triggering terms, avoiding unnecessary quantifier instantiations which would complicate the unsatisfiability proofs.

Relevance. All the bugs we reported to the developers of APRON and ELINA (Chapter 2) and of Z3, CVC4, and MT-ABC (Chapter 3) were considered relevant. Moreover, as our experiments showed, our techniques revealed also bugs reported by other users. We believe this is mainly due to the fact that we use common configuration options, which lead to realistic errors. The benchmarks with known triggering issues we have collected in Chapter 4 contain either serious unsoundnesses which have been fixed or prior incompleteness cases the developers of Viper included in their test suite, to ensure they are not reintroduced by regressions.

Fast error localization. Most of the errors we reported in Chapter 2 and Chapter 3 were confirmed or fixed within days, the developers of MT-ABC explicitly saying that our simple test cases enabled them to understand the root cause of the issues. Moreover, we reported problematic inputs together with the properties expected to hold (in Chapter 2) or with the known ground truth and a possible model or the expected unsat core (in Chapter 3), which we believe contributed to faster error localization. The Gobra developers also confirmed that once they knew which are the required triggering terms (Chapter 4), fixing the unsoundness was considerably faster than manually debugging the axiomatizations. We drew a similar conclusion from the issue tracker of F*, where two developers were initially required to localize an error among 2 388 axioms. We believe the triggering term our algorithm found in ≈60 s would have significantly accelerated this process.

Usability. All our techniques are currently implemented as external tools; integrating them into the development process is mostly an engineering effort.

Applicability. In Chapter 2 and Chapter 3, we have shown that our techniques generalize beyond their initial purpose: the approach for testing classical numerical domains is also effective in finding issues in learning-based transformers; the technique for generating SMT formulas with string operations can also produce formulas that include regular expressions, operations from other theories and their combinations, as well as formulas with soft constraints for testing MAX-SMT solvers. Other possible applications (also for Chapter 4) are discussed in Section 5.2.

5.2 FUTURE WORK

Next, we discuss various directions in which our work could be extended and the challenges that might need to be addressed. Our techniques were designed at the intersection of programming languages and software engineering, but they could also incorporate approaches from other research areas, such as machine learning, program synthesis, program repair, and education technologies. Table 5.1 presents an overview of the possible extensions; the details follow below. Future work specific to a particular type of program analysis tool is presented in Section 5.2.1 (for static analyzers, based on Table 5.1, column 1), Section 5.2.2 (for SMT solvers, as described in Table 5.1, column 2), and Section 5.2.3 (for SMT-based program verifiers, according to Table 5.1, column 3). Section 5.2.4 summarizes open research problems applicable to multiple types of tools (listed in Table 5.1, column 4).

Possible extensions			
Static analyzers	**SMT solvers**	**SMT-based program verifiers**	**Program analysis tools**
string & heap domains (**PL**)	clustering (**ML**)	avoiding matching loops (**PL**)	generic framework (**SE**)
NNs verification domains (**ML**)	generating ⇔ (**ML&PSyn**)	extended unification (**PL&ML**)	non-expert users (**PSyn**)
benchmark programs (**SE**)	E-matching & MBQI (**SE**)	best configuration (**ML**)	fixing the errors (**PR&ML**)
	other provers (**SE**)	new QI algorithm (**PL&SE**)	exam questions (**EdTech**)
		triggering semantics (**PL**)	

PL = programming languages	**ML** = machine learning	**PSyn** = program synthesis
SE = software engineering	**EdTech** = education technologies	**PR** = program repair
NNs = neural networks	⇔ = equivalent formulas/equivalences	QI = quantifier instantiation

TABLE 5.1: Overview of possible extensions of our work across different research areas.

5.2.1 *Static analyzers*

In the following, we explain possible extensions for our approach from Chapter 2.

String and heap domains. Our work focuses on numerical abstract domains, but its main ingredients might carry over to other domains, such as string or heap domains (for testing points-to analyzers). However, these domains may require different techniques to construct the domain elements, as well as suitable parameters for assignments and conditionals, since fuzzers may not be able to effectively explore the search space of non-numerical domain implementations.

Domains for neural networks verification. We have experimentally shown that our technique is also applicable to domains that combine abstract interpretation with machine learning. Recently, there have been also proposed several abstract domains specifically designed for verifying neural networks that support ReLU, Sigmoid, and Tanh activation functions, such as DeepZ [145] (which contains pointwise Zonotope transformers) and DeepPoly [146] (which combines floating-point Polyhedra with Intervals). Since these domains are integrated into ELINA, our approach can be easily extended to test them as well.

Benchmark programs. Our technique performs unit testing, thus it only checks the implementation of domain operations, which are critical parts of a static analyzer. However, this approach may miss errors in other components (e.g., which parse the source code, transform it into intermediate representations, such as abstract syntax trees or control flow graphs, etc.). An alternative would be to consider the analyzer as a whole and to generate benchmark programs as inputs. To facilitate error localization, these could have increasing complexity and contain different classes of issues (e.g., they could throw, by construction, a particular type of exception). We explored this research direction together with Hurmuz, in her Master thesis [89], for null pointer analysis in Java. Our experimental results (see Chapter 6.2 from [89]) show that this approach can effectively identify soundness errors in Infer [64], a widely-used static analyzer. We believe that our work can be further generalized to other types of analysis (e.g., division by zero, buffer overrun, etc.).

5.2.2 *SMT solvers*

Next, we present possible future extensions for our technique from Chapter 3.

Clustering. Multiple failed tests may have the same root cause. To be able to match them with known issues, we manually inspected and grouped some of the failed tests by their common reason. To reduce the human effort, one could cluster the failed tests and characterize the inputs belonging to the cluster automatically (e.g., all of them contain `indexOf` operations with negative offsets). One may also try to express these conditions as regular expressions.

Generating equivalent formulas. Our approach relies on equivalent formulas and equivalences for variables and constants, which are currently written manually. However, this process could be automated by using machine learning or synthesis techniques, as in the works of Singh and Solar-Lezama [150] or Nötzli et al. [128].

Testing E-matching and MBQI. Another research direction would be to automatically test the quantifier instantiation mechanism of an SMT solver, i.e., its two main algorithms, E-matching and MBQI. While our unsat formulas with quantifiers were designed for E-matching, our technique does not provide any guarantees that they contain all the necessary triggering terms for the proof to succeed. It would thus be necessary to extend our approach to also know, by construction, if a formula is expected to be solved by a particular algorithm.

Other provers. Our synthesized formulas with known ground truth could be also used for testing other provers that accept SMT inputs (e.g., Vampire) and decision procedures with machine-checkable proofs (i.e., formalized in Coq, as in [87]).

5.2.3 *SMT-based program verifiers*

Next, we discuss future research directions for the work presented in Chapter 4.

Avoiding matching loops. Our experimental results showed that our synthesized triggering terms can sometimes trigger matching loops. To improve the performance of our algorithm, we could thus identify the (combinations of) triggering terms that cause the validation step to time out and avoid them in future iterations.

Extended unification. Our simplified unification identifies rewritings only when the outermost function is uninterpreted (this considerably reduces their number). An extended version, which also unifies outermost interpreted functions, could be more precise, but we believe much more inefficient (especially for axiomatizations with a few, user-defined types, where only equality would result in a significant number of possible rewritings). For an efficient extended unification, one could first identify, from the previous fixes, which of the arguments of an interpreted function are more error-prone. These could then be prioritized in the rewritings.

Best configuration. We evaluated our algorithm on a few, manually-defined configurations. However, other configurations may have been more effective for par-

ticular benchmarks. Since the space of all combinations of parameters is very large, our approach could be extended to automatically identify the best combination for a given input (or to apply various configurations, in a portfolio manner).

New quantifier instantiation algorithm. Our work could also serve as a starting point for developing a new, E-matching-based, quantifier instantiation algorithm. Integrating it into an SMT solver could also improve its performance (for example, our tool identifies simple, syntactic conflicts between instantiations; these can be automatically and more efficiently detected by a solver during pre-processing).

Encoding the triggering semantics. Algorithms such as MBQI or first-order refutation techniques (available in Vampire) proved to be effective in generating unsatisfiability proofs. However, they do not consider patterns, thus they can produce unsound results for those axiomatizations which are, by construction, only sound module patterns. To benefit from these approaches (and to allow SMT-based verifiers also run in a portfolio mode), we believe it is possible to encode the triggering semantics in SMT, and thus to prevent those instantiations which would be performed by alternative algorithms, but not by E-matching.

5.2.4 *Program analysis tools*

The following possible extensions apply to more than one kind of analysis tool.

Generic framework. Our work is not the only one targeting program analysis tools. There have been significant research efforts (summarized in Chapter 2) in automatically identifying errors in compilers, debuggers, DSE engines, model checkers, SMT solvers, and static analyzers. While all proposed approaches showed improvements over the state-of-the-art, there is no definite solution. We, therefore, believe that based on the strengths and weaknesses of each technique, it is possible to build a generic framework for testing program analysis tools that unifies (some of) the existing approaches. The framework will then select the combination of techniques that is more likely to be effective for the particular tool under test.

Non-expert users. Despite the increased effort of the research community in making formal methods in general, and program analysis tools in particular, more accessible to a wider audience, their usage still requires expert knowledge. However, our techniques allow one to understand which parts are more challenging to implement correctly (i.e., are more error-prone). The approaches presented in this dissertation could be extended to capture the user intent and provide support for automatic code generation: e.g., they could help less experienced developers implement sound join transformers or decision procedures for string concatenation.

Fixing the errors. Another extension would be to automatically localize the source of the errors in the implementation, based on our test cases or synthesized triggering terms, and to suggest a possible fix. To obtain realistic fixes, one could learn this information from the code changes made by the developers.

Exam questions and grading. Our proposed techniques could be used to automatically generate exam questions, following Hozzova et al.'s line of work [88]. For example, for a course on Program Verification that presents the E-matching algorithm, the students could be given an inconsistent axiomatization together with the required triggering terms and they could be asked to explain how E-matching proves unsat. The Axiom Profiler [19] can automatically generate a possible solution for this problem. Moreover, our test cases from Chapter 2 could be used to automatically grade a project for a Program Analysis course, where the students are required to implement their own, simplified, static analyzer.

BIBLIOGRAPHY

1. Ahn, K. Y. & Denney, E. *Testing First-Order Logic Axioms in Program Verification* in *Tests and Proofs* (eds Fraser, G. & Gargantini, A.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010), 22.
2. *Amazon Web Services* `https://aws.amazon.com/`.
3. Amighi, A., Blom, S. & Huisman, M. *VerCors: A Layered Approach to Practical Verification of Concurrent Software* in *PDP* (IEEE Computer Society, 2016), 495.
4. Andreasen, E. S., Møller, A. & Nielsen, B. B. *Systematic Approaches for Increasing Soundness and Precision of Static Analyzers* in *SOAP* (ACM, 2017), 31.
5. *Array Maximum, by elimination* `http://viper.ethz.ch/examples/max-array-elimination.html`. 2021.
6. Artho, C., Biere, A. & Seidl, M. *Model-Based Testing for Verification Back-Ends* in *Tests and Proofs* (eds Veanes, M. & Viganò, L.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013), 39.
7. Astrauskas, V., Müller, P., Poli, F. & Summers, A. J. *Leveraging Rust Types for Modular Specification and Verification* in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* **3** (ACM, 2019), 147:1.
8. Aydin, A., Bang, L. & Bultan, T. *Automata-Based Model Counting for String Constraints* in *Computer Aided Verification* (eds Kroening, D. & Pasareanu, C. S.) (Springer International Publishing, Cham, 2015), 255.
9. Aydin, A., Eiers, W., Bang, L., Brennan, T., Gavrilov, M., Bultan, T. & Yu, F. *Parameterized Model Counting for String and Numeric Constraints* in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ACM, Lake Buena Vista, FL, USA, 2018), 400.
10. Baader, F. & Snyder, W. *Unification Theory* in *Handbook of Automated Reasoning* (eds Robinson, J. A. & Voronkov, A.) (Elsevier and MIT Press, 2001), 445.
11. Backes, J. D., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K. S., Rungta, N., Tkachuk, O. & Varming, C. Semantic-based Automated Reasoning for AWS Access Policies using SMT. *2018 Formal Methods in Computer Aided Design (FMCAD)*, 1 (2018).
12. Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C. & Finocchi, I. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* **51** (2018).
13. Ball, T. & Rajamani, S. K. *The SLAM Project: Debugging System Software via Static Analysis* in *POPL* (ACM, 2002), 1.
14. Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B. & Leino, K. R. M. *Boogie: A modular reusable verifier for object-oriented programs* in *Formal Methods for Components and Objects (FMCO)* (eds de Boer, F. S., Bonsangue, M. M., Graf, S. & de Roever, W. P.) **5** (Springer, 2005), 364.
15. Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W. & Venter, H. Specification and Verification: The Spec# Experience. *Communications of the ACM* **54**, 81 (2011).

16. Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. & Yoo, S. The Oracle Problem in Software Testing: A Survey. *TSE* **41**, 507 (2015).

17. Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A. & Tinelli, C. *CVC4* in *Computer Aided Verification* (eds Gopalakrishnan, G. & Qadeer, S.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011), 171.

18. Barrett, C., Fontaine, P. & Tinelli, C. *The SMT-LIB Standard: Version 2.6* tech. rep. Available at http://www.SMT-LIB.org (Department of Computer Science, The University of Iowa, 2017).

19. Becker, N., Müller, P. & Summers, A. J. *The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations* in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (eds Vojnar, T. & Zhang, L.) **11427** (Springer-Verlag, 2019), 99.

20. Becker, O. *Automatically Testing SMT Solvers* Bachelor Thesis (ETH Zurich, Switzerland, 2021).

21. Berzish, M., Ganesh, V. & Zheng, Y. *Z3str3: A String Solver with Theory-aware Heuristics* in *2017 Formal Methods in Computer Aided Design (FMCAD)* (2017), 55.

22. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J. D., Nowotka, D. & Ganesh, V. *An SMT Solver for Regular Expressions and Linear Arithmetic over String Length* in *Computer Aided Verification* (eds Silva, A. & Leino, K. R. M.) (Springer International Publishing, Cham, 2021), 289.

23. Beyer, D., Henzinger, T. A., Keremoglu, M. E. & Wendler, P. *Conditional Model Checking: A Technique to Pass Information between Verifiers* in *FSE* (ACM, 2012), 57.

24. Beyer, D. & Keremoglu, M. E. *CPAchecker: A Tool for Configurable Software Verification* in *CAV* **6806** (Springer, 2011), 184.

25. Bielik, P., Fischer, M. & Vechev, M. Robust Relational Layout Synthesis from Examples for Android. *Proc. ACM Program. Lang.* **2** (2018).

26. Bielik, P., Raychev, V. & Vechev, M. T. *Learning a Static Analyzer from Data* in *CAV (1)* (2017), 233.

27. Biere, A., Heule, M., van Maaren, H. & Walsh, T. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications* (IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009).

28. Bjørner, N., Ganesh, V., Michel, R. & Veanes, M. An SMT-LIB Format for Sequences and Regular Expressions. *Strings* (2012).

29. Bjørner, N., Phan, A.-D. & Fleckenstein, L. *νZ - An Optimizing SMT Solver* in *TACAS* (2015).

30. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D. & Rival, X. *A Static Analyzer for Large Safety-critical Software* in *PLDI* (ACM, 2003), 196.

31. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I. & Ganesh, V. *StringFuzz: A Fuzzer for String Solvers* in *Computer Aided Verification* (eds Chockler, H. & Weissenbacher, G.) (Springer International Publishing, Cham, 2018), 45.

32. Böhme, S. & Weber, T. *Fast LCF-Style Proof Reconstruction for Z3* in *Interactive Theorem Proving* (eds Kaufmann, M. & Paulson, L. C.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010), 179.

33. Boyapati, C., Khurshid, S. & Marinov, D. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Softw. Eng. Notes* **27**, 123 (2002).

34. Brummayer, R. & Biere, A. Fuzzing and delta-debugging SMT solvers. *ACM International Conference Proceeding Series*, 1 (2009).

35. Brummayer, R., Lonsing, F. & Biere, A. *Automated Testing and Debugging of SAT and QBF Solvers* in *Theory and Applications of Satisfiability Testing – SAT 2010* (eds Strichman, O. & Szeider, S.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010), 44.

36. Bugariu, A., Ter-Gabrielyan, A. & Müller, P. *Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers (extended version)* tech. rep. 2105.04385 (arXiv, 2021).

37. Bugariu, A. & Müller, P. *Automatically Testing String Solvers* in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Association for Computing Machinery, Seoul, South Korea, 2020), 1459.

38. Bugariu, A., Ter-Gabrielyan, A. & Müller, P. *Identifying Overly Restrictive Matching Patterns in SMT-Based Program Verifiers* in *Formal Methods* (eds Huisman, M., Pasareanu, C. & Zhan, N.) (Springer International Publishing, Cham, 2021), 273.

39. Bugariu, A., Wüstholz, V., Christakis, M. & Müller, P. *Automatically Testing Implementations of Numerical Abstract Domains* in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Association for Computing Machinery, Montpellier, France, 2018), 768.

40. Cadar, C. & Donaldson, A. F. *Analysing the Program Analyser* in *Proceedings of the 38th International Conference on Software Engineering Companion* (ACM, Austin, Texas, 2016), 765.

41. Cadar, C., Dunbar, D. & Engler, D. R. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs* in *OSDI* (USENIX, 2008), 209.

42. Cadar, C. & Engler, D. R. *Execution Generated Test Cases: How to Make Systems Code Crash Itself* in *SPIN* **3639** (Springer, 2005), 2.

43. Cadar, C. & Sen, K. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* **56**, 82 (2013).

44. Casso, I., Morales, J. F., López-García, P. & Hermenegildo, M. V. *Testing Your (Static Analysis) Truths* in *Logic-Based Program Synthesis and Transformation* (ed Fernández, M.) (Springer International Publishing, Cham, 2021), 271.

45. Chatterjee, S., Lahiri, S. K., Qadeer, S. & Rakamarić, Z. *A Reachability Predicate for Analyzing Low-Level Software* in *Tools and Algorithms for the Construction and Analysis of Systems* (eds Grumberg, O. & Huth, M.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2007), 19.

46. Christakis, M., Müller, P. & Wüstholz, V. *An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer* in *VMCAI* **8931** (Springer, 2015), 336.

47. Christakis, M., Müller, P. & Wüstholz, V. *Collaborative Verification and Testing with Explicit Assumptions* in *FM* **7436** (Springer, 2012), 132.

48. Christakis, M. & Wüstholz, V. *Bounded Abstract Interpretation* in *SAS* **9837** (Springer, 2016), 105.

49. Claessen, K. & Hughes, J. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs* in *ICFP* (ACM, 2000), 268.
50. *Clang: a C language family frontend for LLVM* `https://clang.llvm.org/`.
51. Cousot, P. & Cousot, R. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints* in *POPL* (ACM, 1977), 238.
52. Cousot, P. & Cousot, R. *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation* in *PLILP* **631** (Springer, 1992), 269.
53. Cousot, P. & Cousot, R. *Static Determination of Dynamic Properties of Programs* in *ISOP* (Dunod, 1976), 106.
54. Cousot, P. & Cousot, R. *Systematic Design of Program Analysis Frameworks* in *POPL* (ACM, 1979), 269.
55. Cousot, P. & Halbwachs, N. *Automatic Discovery of Linear Restraints Among Variables of a Program* in *POPL* (ACM, 1978), 84.
56. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B. & Yang, X. *Testing Static Analyzers with Randomly Generated Programs* in *NFM* **7226** (Springer, 2012), 120.
57. *CVC4 Documentation for the String Theory* `http://cvc4.cs.stanford.edu/wiki/Strings`.
58. *CVC4 Regression Test Suite* `https://github.com/CVC4/CVC4/tree/master/test/regress`.
59. *CVC5 SMT Solver* `https://cvc5.github.io/`.
60. Darulova, E. & Kuncak, V. *Sound Compilation of Reals* in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (ACM, San Diego, California, USA, 2014), 235.
61. Darvas, Á. & Leino, K. R. M. *Practical reasoning about invocations and implementations of pure methods* in *Fundamental Approaches to Software Engineering (FASE)* (eds Dwyer, M. B. & Lopes, A.) **4422** (Springer-Verlag, 2007), 336.
62. De Moura, L. & Bjørner, N. *Efficient E-Matching for SMT Solvers* in *Automated Deduction – CADE-21* (ed Pfenning, F.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2007), 183.
63. Detlefs, D., Nelson, G. & Saxe, J. B. Simplify: A Theorem Prover for Program Checking. *J. ACM* **52**, 365 (2005).
64. Distefano, D., Fähndrich, M., Logozzo, F. & O'Hearn, P. W. Scaling Static Analyses at Facebook. *Commun. ACM* **62**, 62 (2019).
65. Dubois, C. *Proving ML Type Soundness Within Coq* in *TPHOLs* **1869** (Springer, 2000), 126.
66. Eilers, M. & Müller, P. *Nagini: A Static Verifier for Python* in *Computer Aided Verification (CAV)* (eds Chockler, H. & Weissenbacher, G.) **10982** (Springer International Publishing, 2018), 596.
67. *ELINA Artifact (POPL 2017)* `https://www.sri.inf.ethz.ch/optpoly.php`.
68. *ELINA Artifact (POPL 2018)* `https://www.sri.inf.ethz.ch/popl18-paper251.php`.

69. *ELINA Code Base* `https://github.com/eth-sri/ELINA`.

70. *ELINA LAIT Fixes* `https://github.com/eth-sri/ELINA/commit/51f4e8102904e60ee7e604d71d99aa4443085180`, `https://github.com/eth-sri/ELINA/commit/5d0e0159c8c0ef400ed31b32219ca97a3eacc491`.

71. *ELINA LAIT Tested Version* `https://github.com/eth-sri/ELINA/commit/8400beb7a55087aa7cacdfae2fe851bcc3b27b3d`.

72. *F* Issue 1848* `https://github.com/FStarLang/FStar/issues/1848`. 2021.

73. Fähndrich, M. & Logozzo, F. *Static Contract Checking with Abstract Interpretation* in *FoVeOOS* **6528** (Springer, 2010), 10.

74. Filliâtre, J. & Paskevich, A. *Why3—Where Programs Meet Provers* in *Programming Languages and Systems (ESOP)* (eds Felleisen, M. & Gardner, P.) **7792** (Springer, 2013), 125.

75. Ford, J. & Shankar, N. *Formal Verification of a Combination Decision Procedure* in *Proceedings of the 18th International Conference on Automated Deduction* (Springer-Verlag, Berlin, Heidelberg, 2002), 347.

76. Fu, Z. & Su, Z. in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II* (eds Chaudhuri, S. & Farzan, A.) 187 (Springer International Publishing, Cham, 2016).

77. Gario, M. & Micheli, A. *PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms* in *SMT Workshop 2015* (2015).

78. Ge, Y. & de Moura, L. *Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories* in *Computer Aided Verification* (eds Bouajjani, A. & Maler, O.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009), 306.

79. Ghorbal, K., Goubault, E. & Putot, S. *The Zonotope Abstract Domain Taylor1+* in *CAV* **5643** (Springer, 2009), 627.

80. Godefroid, P., Klarlund, N. & Sen, K. *DART: Directed Automated Random Testing* in *PLDI* (ACM, 2005), 213.

81. Godefroid, P., Levin, M. Y. & Molnar, D. A. *Automated Whitebox Fuzz Testing* in *NDSS* (2008).

82. Goubault, E. & Putot, S. *Static Analysis of Numerical Algorithms* in *SAS* **4134** (Springer, 2006), 18.

83. He, J., Singh, G., Püschel, M. & Vechev, M. *Learning Fast and Precise Numerical Analysis* in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, London, UK, 2020), 1112.

84. He, J., Singh, G., Püschel, M. & Vechev, M. *Reproduction Package for Article: Learning Fast and Precise Numerical Analysis*

85. Henry, J., Monniaux, D. & Moy, M. PAGAI: A Path Sensitive Static Analyser. *Electr. Notes Theor. Comput. Sci.* **289**, 15 (2012).

86. Heule, S., Kassios, I. T., Müller, P. & Summers, A. J. *Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions* in *European Conference on Object-Oriented Programming (ECOOP)* (ed Castagna, G.) **7920** (Springer, 2013), 451.

87. Hooimeijer, P. & Weimer, W. *A Decision Procedure for Subset Constraints over Regular Languages* in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, Dublin, Ireland, 2009), 188.

88. Hozzová, P., Kovács, L. & Rath, J. *Automated Generation of Exam Sheets for Automated Deduction* in *Intelligent Computer Mathematics* (eds Kamareddine, F. & Sacerdoti Coen, C.) (Springer International Publishing, Cham, 2021), 185.

89. Hurmuz, M. *Automatically Generating Java Benchmarks with Known Errors* Master Thesis (ETH Zurich, Switzerland, 2022).

90. Jeannet, B. & Miné, A. *Apron: A Library of Numerical Abstract Domains for Static Analysis* in *CAV* **5643** (Springer, 2009), 661.

91. Jensen, S. H., Møller, A. & Thiemann, P. *Type Analysis for JavaScript* in *Proc. 16th International Static Analysis Symposium (SAS)* **5673** (Springer-Verlag, 2009).

92. Jeon, J., Qiu, X., Fetter-Degges, J., Foster, J. S. & Solar-Lezama, A. *Synthesizing Framework Models for Symbolic Execution* in *Proceedings of the 38th International Conference on Software Engineering* (ACM, Austin, Texas, 2016), 156.

93. Jourdan, J.-H., Laporte, V., Blazy, S., Leroy, X. & Pichardie, D. *A Formally-Verified C Static Analyzer* in *POPL* (ACM, 2015), 247.

94. Kapus, T. & Cadar, C. *Automatic testing of symbolic execution engines via program generation and differential testing* in *ASE* (IEEE Computer Society, 2017), 590.

95. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J. & Yakobowski, B. Frama-C: A software analysis perspective. *Formal Aspects of Computing* **27**, 573 (2015).

96. *KLEE Tutorial* `http://klee.github.io/tutorials/testing-regex/`.

97. Klinger, C., Christakis, M. & Wüstholz, V. *Differentially Testing Soundness and Precision of Program Analyzers* in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Association for Computing Machinery, Beijing, China, 2019), 239.

98. Kovács, L. & Voronkov, A. *First-Order Theorem Proving and Vampire* in *Computer Aided Verification* (eds Sharygina, N. & Veith, H.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013), 1.

99. Kühne, S. *Automatically Testing Solvers for String and Regular Expressions Constraints* Bachelor Thesis (ETH Zurich, Switzerland, 2022).

100. Lal, A. & Qadeer, S. *Powering the Static Driver Verifier Using Corral* in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Association for Computing Machinery, Hong Kong, China, 2014), 202.

101. Lal, A., Qadeer, S. & Lahiri, S. K. *A Solver for Reachability Modulo Theories* in *Computer Aided Verification* (eds Madhusudan, P. & Seshia, S. A.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012), 427.

102. Lattner, C. & Adve, V. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation* in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, 2004).

103. Le, V., Afshari, M. & Su, Z. *Compiler validation via equivalence modulo inputs* in *PLDI* (ACM, 2014), 216.

104. Le, V., Sun, C. & Su, Z. *Finding deep compiler bugs via guided stochastic program mutation* in *OOPSLA* (ACM, 2015), 386.

105. Leavens, G. T., Baker, A. L. & Ruby, C. *JML: a Java Modeling Language* in *In Formal Underpinnings of Java Workshop (at OOPSLA'98* (1998).

106. Leino, K. R. M. *Dafny: An Automatic Program Verifier for Functional Correctness* in *LPAR* **6355** (Springer, 2010), 348.

107. Leino, K. R. M. & Monahan, R. *Reasoning about Comprehensions with First-Order SMT Solvers* in *Proceedings of the 2009 ACM Symposium on Applied Computing* (Association for Computing Machinery, Honolulu, Hawaii, 2009), 615.

108. Leino, K. R. M. & Müller, P. *Verification of Equivalent-Results Methods* in *European Symposium on Programming (ESOP)* (ed Drossopoulou, S.) **4960** (Springer-Verlag, 2008), 307.

109. Leino, K. R. M. & Rümmer, P. *A Polymorphic Intermediate Verification Language: Design and Logical Encoding* in *Tools and Algorithms for the Construction and Analysis of Systems* (eds Esparza, J. & Majumdar, R.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010), 312.

110. Leroy, X. Formal verification of a realistic compiler. *CACM* **52**, 107 (2009).

111. Lescuyer, S. & Conchon, S. *A Reflexive Formalization of a SAT Solver in Coq* in *In Proceedings of TPHOLs* (2008).

112. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C. & Deters, M. An Efficient SMT Solver for String Constraints. *Form. Methods Syst. Des.* **48**, 206 (2016).

113. *LibFuzzer—A Library for Coverage-Guided Fuzz Testing* `https://llvm.org/docs/LibFuzzer.html`.

114. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A. & Vardoulakis, D. In Defense of Soundiness: A Manifesto. *CACM* **58**, 44 (2 2015).

115. Logozzo, F. & Fähndrich, M. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.* **75**, 796 (2010).

116. Madsen, M. & Lhoták, O. *Safe and Sound Program Analysis with FLIX* in *ISSTA* To appear (ACM, 2018).

117. Mansur, M. N., Christakis, M., Wüstholz, V. & Zhang, F. *Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing* in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Virtual Event, USA, 2020), 701.

118. Mari, F. Formal Verification of a Modern SAT Solver by Shallow Embedding into Isabelle/HOL. *Theor. Comput. Sci.* **411**, 4333 (2010).

119. McKeeman, W. M. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* **10**, 100 (1998).

120. Midtgaard, J. & Møller, A. QuickChecking Static Analysis Properties. *Softw. Test. Verif. Reliab.* **27** (2017).

121. Miné, A. The Octagon Abstract Domain. *Higher Order Symbol. Comput.* **19**, 31 (2006).

122. Moskal, M. *Programming with Triggers* in *SMT* **375** (ACM, 2009), 20.

123. *MT-ABC Tested Version* `https://github.com/vlab-cs-ucsb/ABC/commit/86b00141fddd183de7b9ae5c92c240e19dda1950`.

124. Müller, P., Schwerhoff, M. & Summers, A. J. *Viper: A Verification Infrastructure for Permission-Based Reasoning* in *Verification, Model Checking, and Abstract Interpretation (VMCAI)* (eds Jobstmann, B. & Leino, K. R. M.) **9583** (Springer-Verlag, 2016), 41.

125. Muske, T. & Serebrenik, A. *Survey of Approaches for Handling Static Analysis Alarms* in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2016), 157.

126. Niemetz, A., Preiner, M. & Biere, A. *Model-Based API Testing for SMT Solvers* in *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017), affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017* (eds Brain, M. & Hadarean, L.) (2017), 10 pages.

127. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C. & Tinelli, C. *Towards Bit-Width-Independent Proofs in SMT Solvers* in *Automated Deduction – CADE 27* (ed Fontaine, P.) (Springer International Publishing, Cham, 2019), 366.

128. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C. & Tinelli, C. *Syntax-Guided Rewrite Rule Enumeration for SMT Solvers* in *Theory and Applications of Satisfiability Testing – SAT 2019* (eds Janota, M. & Lynce, I.) (Springer International Publishing, Cham, 2019), 279.

129. *Our generated benchmarks from the string theory* `http://github.com/alebugariu/StringSolversTests/tree/master/experiments/generatedTests`.

130. Pacheco, C., Lahiri, S. K., Ernst, M. D. & Ball, T. *Feedback-Directed Random Test Generation* in *ICSE* (IEEE Computer Society, 2007), 75.

131. Park, J. *SMT Solver Testing with Type and Grammar Based Mutation* in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Athens, Greece, 2021), 1675.

132. Park, J., Winterer, D., Zhang, C. & Su, Z. Generative Type-Aware Mutation for Testing SMT Solvers. *Proc. ACM Program. Lang.* (2021).

133. Paulin-Mohring, C. in *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures* (eds Meyer, B. & Nordio, M.) 45 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012).

134. Reger, G., Bjorner, N., Suda, M. & Voronkov, A. *AVATAR Modulo Theories* in *GCAI 2016. 2nd Global Conference on Artificial Intelligence* (eds Benzmüller, C., Sutcliffe, G. & Rojas, R.) **41** (EasyChair, 2016), 39.

135. Reps, T. W., Sagiv, S. & Yorsh, G. *Symbolic Implementation of the Best Transformer* in *VMCAI* **2937** (Springer, 2004), 252.

136. Reynolds, A., Barbosa, H. & Fontaine, P. *Revisiting Enumerative Instantiation* in *Tools and Algorithms for the Construction and Analysis of Systems* (eds Beyer, D. & Huisman, M.) (Springer International Publishing, Cham, 2018), 112.

137. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C. & Barrett, C. *Counterexample-Guided Quantifier Instantiation for Synthesis in SMT* in *Computer Aided Verification* (eds Kroening, D. & Pasareanu, C. S.) (Springer International Publishing, Cham, 2015), 198.

138. Rudich, A., Darvas, Á. & Müller, P. *Checking Well-Formedness of Pure-Method Specifications* in *Formal Methods (FM)* (eds Cuellar, J. & Maibaum, T.) **5014** (Springer-Verlag, 2008), 68.

139. Rümmer, P. *E-Matching with Free Variables* in *Logic for Programming, Artificial Intelligence, and Reasoning* (eds Bjørner, N. & Voronkov, A.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012), 359.

140. Schulte, W. *VCC: Contract-based Modular Verification of Concurrent C* in *31st International Conference on Software Engineering, ICSE 2009* (IEEE Computer Society, 2008).

141. Scott, J., Mora, F. & Ganesh, V. *BanditFuzz: A Reinforcement-Learning Based Performance Fuzzer for SMT Solvers* in *Software Verification* (eds Christakis, M., Polikarpova, N., Duggirala, P. S. & Schrammel, P.) (Springer International Publishing, Cham, 2020), 68.

142. Sen, K. & Agha, G. *CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools* in *CAV* **4144** (Springer, 2006), 419.

143. Sen, K., Marinov, D. & Agha, G. *CUTE: A Concolic Unit Testing Engine for C* in *ESEC/FSE* (ACM, 2005), 263.

144. Shao, Z., Saha, B., Trifonov, V. & Papaspyrou, N. *A type system for certified binaries* in *POPL* (ACM, 2002), 217.

145. Singh, G., Gehr, T., Mirman, M., Püschel, M. & Vechev, M. *Fast and Effective Robustness Certification* in (Curran Associates Inc., Montréal, Canada, 2018), 10825.

146. Singh, G., Gehr, T., Püschel, M. & Vechev, M. An Abstract Domain for Certifying Neural Networks. *Proc. ACM Program. Lang.* **3** (2019).

147. Singh, G., Püschel, M. & Vechev, M. A Practical Construction for Decomposing Numerical Abstract Domains. *PACMPL* **2**, 55:1 (2018).

148. Singh, G., Püschel, M. & Vechev, M. *Fast Polyhedra Abstract Domain* in *POPL* (ACM, 2017), 46.

149. Singh, G., Püschel, M. & Vechev, M. *Making Numerical Program Analysis Fast* in *PLDI* (ACM, 2015), 303.

150. Singh, R. & Solar-Lezama, A. *SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules* in *2016 Formal Methods in Computer-Aided Design (FMCAD)* (2016), 185.

151. *SMT-COMP* `https://smt-comp.github.io`.

152. *SMT-LIB Arrays Theory* `http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml`.

153. *SMT-LIB Booleans Theory* `http://smtlib.cs.uiowa.edu/theories-Core.shtml`.

154. *SMT-LIB Fixed-Size Bit-Vectors Theory* `http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml`.

155. *SMT-LIB Integers Theory* `http://smtlib.cs.uiowa.edu/theories-Ints.shtml`.

156. *SMT-LIB Unicode Strings Theory* `http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml/`.

157. Strebel, P. *Explaining Unsatisfiability Proofs through Examples* Bachelor Thesis (ETH Zurich, 2021).

158. *StringFuzz Test Suite* http://stringfuzz.dmitryblotsky.com/problems/.

159. Stump, A., Oe, D., Reynolds, A., Hadarean, L. & Tinelli, C. SMT Proof Checking Using a Logical Framework. *Form. Methods Syst. Des.* **42**, 91 (2013).

160. Sun, C., Le, V. & Su, Z. *Finding and analyzing compiler warning defects* in *ICSE* (ACM, 2016), 203.

161. Sutcliffe, G. The CADE ATP System Competition - CASC. *AI Magazine* **37**, 99 (2016).

162. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., Zinzindohoue, J.-K. & Zanella-Béguelin, S. *Dependent Types and Multi-Monadic Effects in F** in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Association for Computing Machinery, St. Petersburg, FL, USA, 2016), 256.

163. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J. & Livshits, B. *Verifying Higher-order Programs with the Dijkstra Monad* in *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation* (2013), 387.

164. T. Y. Chen, S. C. C. & Yiu, S.-M. *Metamorphic testing: a new approach for generating next test cases* in *Technical Report HKUST-CS98-01* (1998).

165. Taneja, J., Liu, Z. & Regehr, J. *Testing Static Analyses for Precision and Soundness* in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (Association for Computing Machinery, San Diego, CA, USA, 2020), 81.

166. *Technical "Whitepaper" for AFL* http://lcamtuf.coredump.cx/afl/technical_details.txt.

167. *The 14th International Satisfiability Modulo Theories Competition (including pending benchmarks)* https://smt-comp.github.io/2019/,https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks-tmp/benchmarks-pending. 2019.

168. *The 15th International Satisfiability Modulo Theories Competition* https://smt-comp.github.io/2020/. 2020.

169. *The APRON Library Documentation* http://apron.cri.ensmp.fr/library/0.9.10/apron.pdf.

170. *The dk.brics.automaton Package: Finite-state automaton for regular expressions* https://www.brics.dk/automaton/.

171. *The Jazzer coverage-guided fuzzer* https://github.com/CodeIntelligenceTesting/jazzer.

172. Thommen, K. *Automatically Testing MAX-SMT Solvers* Bachelor Thesis (ETH Zurich, Switzerland, 2022, to appear).

173. Tillmann, N. & De Halleux, J. *Pex: White Box Test Generation for .NET* in *Proceedings of the 2nd International Conference on Tests and Proofs* (Springer-Verlag, Prato, Italy, 2008), 134.

174. Tolksdorf, S., Lehmann, D. & Pradel, M. *Interactive Metamorphic Testing of Debuggers* in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Association for Computing Machinery, Beijing, China, 2019), 273.

175. Verge, H. L. *A note on Chernikova's Algorithm* tech. rep. RR-1662 (INRIA, 1992).

176. *Viper test suite* `https://github.com/viperproject/silver/tree/master/src/test/resources`. 2021.

177. Vishwanathan, H., Shachnai, M., Narayana, S. & Nagarakatte, S. *Semantics, Verification, and Efficient Implementations for Tristate Numbers* 2021.

178. Visser, W., Păsăreanu, C. S. & Khurshid, S. Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes* **29**, 97 (2004).

179. Voronkov, A. *AVATAR: The Architecture for First-Order Theorem Provers* in *Computer Aided Verification* (eds Biere, A. & Bloem, R.) (Springer International Publishing, Cham, 2014), 696.

180. Wei, S., Mardziel, P., Ruef, A., Foster, J. S. & Hicks, M. *Evaluating Design Tradeoffs in Numeric Static Analysis for Java* in *ESOP* **10801** (Springer, 2018), 653.

181. Winterer, D., Zhang, C. & Su, Z. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* **4** (2020).

182. Winterer, D., Zhang, C. & Su, Z. *Validating SMT Solvers via Semantic Fusion* in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, London, UK, 2020), 718.

183. Wolf, F. A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J. C. & Müller, P. *Gobra: Modular Specification and Verification of Go Programs* in *Computer Aided Verification (CAV)* (eds Silva, A. & Leino, K. R. M.) (Springer International Publishing, 2021), 367.

184. Yang, X., Chen, Y., Eide, E. & Regehr, J. *Finding and understanding bugs in C compilers* in *PLDI* (ACM, 2011), 283.

185. Yao, P., Huang, H., Tang, W., Shi, Q., Wu, R. & Zhang, C. *Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration* in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Association for Computing Machinery, Virtual, Denmark, 2021), 322.

186. Yao, P., Huang, H., Tang, W., Shi, Q., Wu, R. & Zhang, C. *Skeletal Approximation Enumeration for SMT Solver Testing* in (Association for Computing Machinery, Athens, Greece, 2021), 1141.

187. *Z3 SMT Solver* `https://github.com/Z3Prover/z3/`.

188. *Z3 Test Suite* `https://github.com/Z3Prover/z3/tree/master/src/test`.

189. Zeller, A. & Hildebrandt, R. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* **28**, 183 (2002).

190. Zhang, C. *Stress Testing SMT Solvers via Type-aware Mutation* in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2020), 119.

191. Zhang, C., Su, T., Yan, Y., Zhang, F., Pu, G. & Su, Z. *Finding and Understanding Bugs in Software Model Checkers* in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Tallinn, Estonia, 2019), 763.

192. Zhang, L. & Malik, S. *Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications* in *2003 Design, Automation and Test in Europe Conference and Exhibition* (2003), 880.

193. Zhang, Q., Sun, C. & Su, Z. *Skeletal Program Enumeration for Rigorous Compiler Testing* in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, Barcelona, Spain, 2017), 347.

194. Zheng, Y., Zhang, X. & Ganesh, V. *Z3-str: A Z3-based String Solver for Web Application Analysis* in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (ACM, Saint Petersburg, Russia, 2013), 114.

CURRICULUM VITAE

PERSONAL DATA

Name	Alexandra-Olimpia Bugariu
Date of Birth	12.01.1990
Place of Birth	Timisoara, Romania
Citizen of	Romania
Email	alexandra.bugariu@inf.ethz.ch

EDUCATION

2016 – 2022 PhD in Computer Science
ETH Zurich, Switzerland
Advisor: Prof. Dr. Peter Müller
Dissertation: Automatically Identifying Soundness and Completeness Errors in Program Analysis Tools

2013 – 2015 MSc. in Computer Science
(European Master in Software Engineering, Double Degree)
Free University of Bozen-Bolzano, Italy and Technische Universität Kaiserslautern, Germany
Grade: 110 cum laude/110 (Italy) and 1.0/1.0 (Germany)
Thesis: Alternative Approaches for Quantitative State/Event Fault Trees Analysis

2009 – 2013 BSc. in Computer and Software Engineering
(Major in Software Engineering)
Politehnica University of Timisoara, Romania
Grade: 9.67/10
Thesis: Extensions of the phantm Analyzer to support Model Extraction from PHP Web Applications

HONORS & AWARDS

2021 Young Researcher at the 8th Heidelberg Laureate Forum
Selected among 200 young researchers from Mathematics and Computer Science

2020 Young Researcher at the Virtual Heidelberg Laureate Forum
Selected among 200 young researchers from Mathematics and Computer Science

2015 DAAD STIBET Scholarship
3 months scholarship for finalizing my master studies at the Technical University of Kaiserslautern

2013 – 2015 Erasmus Mundus Consortium Scholarship
2 years scholarship for the European Master in Software Engineering

2009 Valedictorian
Award for the high school student with the highest grade (10/10)

EMPLOYMENT

September 2016 – September 2022 Teaching Assistant (TA)
ETH Zurich, Switzerland
BSc./MSc. seminars/exercise sessions for: Informatik I (AS'16), Software Architecture and Engineering (SS'17, SS'18 - Head TA), Concepts of Object-Oriented Programming (AS'17, AS'18, AS'19, AS'20 - Head TA, AS'21 - Head TA), Research Topics in Software Engineering (AS'17, AS'19), Parallel Programming (SS'19), Rigorous Software Engineering (SS'20)

March 2016 – August 2016 Software engineer intern
itemis AG Stuttgart, Germany
Development of formal analysis tools (based on the Z3 SMT solver) for domain specific languages

February 2013 – July 2013 Research intern
e-Austria Institute Timisoara, Romania
Development of formal analysis tools (based on the Yices and Z3 SMT solvers) for systems modeled as state automata, development of my BSc. thesis

August 2012 – September 2012 Research intern
fortiss GmbH München, Germany
Development of formal analysis tools (based on the Yices SMT solver and on the NuSMV model checker) for checking the compatibility of components and the feasibility of message sequence charts

July 2010 – September 2010 Student intern
TRW Automotive Timisoara, Romania
Testing, development of tools for internal use (Word reports generator from XML files using C#)

SERVICE

Scientific staff representative	ETH Zurich Hiring Committee for Computer Science – Security, Software Engineering, and Programming Languages'21
Subreviewer	SEFM'20, ISSTA'19, EMSOFT'18, FM'18
PC	PLDI'22 AEC, CAV'21 AEC, PLDI'21 AEC
Student volunteer	PLDI'19, ECOOP/PLDI'17

STUDENTS ADVISED

AS'21	Sebastian Kühne: Automatically Testing Solvers for String and Regular Expressions Constraints (ETH Zurich, BSc Thesis)
	Kevin Thommen: Automatically Testing MAX-SMT Solvers (ETH Zurich, BSc Thesis)
	Madalina Hurmuz: Automatically Generating Java Benchmarks with Known Errors (ETH Zurich, MSc Thesis)
	Olivier Becker: Automatically Testing SMT Solvers (ETH Zurich, BSc Thesis)
SS'21	Pascal Strebel: Explaining Unsatisfiability Proofs through Examples (ETH Zurich, BSc Thesis, co-advised with Dr. Malte Schwerhoff)
SS'18	Radwa Sherif Abdelbar: Automated Checking of Implicit Assumptions on Textual Data (ETH Zurich, BSc Thesis, co-advised with Dr. Caterina Urban)
AS'17	Madelin Schumacher: Automated Generation of Data Quality Checks (ETH Zurich, MSc Thesis, co-advised with Dr. Caterina Urban)

PUBLICATIONS

[FM'21]	Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers (acceptance rate: 26.7%) Alexandra Bugariu, Arshavir Ter-Gabrielyan, Peter Müller
[ICSE'20]	Automatically Testing String Solvers (acceptance rate: 20.9%) Alexandra Bugariu, Peter Müller
[ASE'18]	Automatically Testing Implementations of Numerical Abstract Domains (acceptance rate: 19.9%) Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, Peter Müller