Thomas Lemberger

Towards Cooperative Software Verification with Test Generation and Formal Verification

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München • Software and Computational Systems Lab • Eingereicht von Thomas Lemberger am 18. Mai 2022

J. Pöpperl
2.11.2003

Thomas Lemberger

Towards Cooperative Software Verification with Test Generation and Formal Verification

1. Gutachter: Prof. Dr. Dirk Beyer
2. Gutachter: Prof. doc. RNDr. Jan Strejček
3. Gutachter: Prof. Jonathan Bell, PhD

Tag der Einreichung: 18. Mai 2022
Tag der mündlichen Prüfung: 12. Dezember 2022

Dissertation an der Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München • Software and Computational Systems Lab

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

Lemberger, Thomas
Name, Vorname

Ingolstadt, 18. Mai 2022	Thomas Lemberger
Ort, Datum	Unterschrift Doktorand/in

Zusammenfassung

Es existieren zwei typische Methoden für die Verifikation von Software: Testen und formale Verifikation. Um unser Vertrauen in Software in der breiten Masse zu erhöhen, benötigen wir Werkzeuge, die diese Methoden automatisch und zuverlässig anwenden. Testen mit manuell geschriebenen Tests ist weit verbreitet, aber es gibt kein standardisiertes Format für automatisch generierte Tests für die Programmiersprache C. Dies macht die Verwendung und den Vergleich von automatischen Testgeneratoren aufwendig. Darüber hinaus können Tests niemals volles Vertrauen in Software bieten — sie können das Vorhandensein von Programmfehlern nachweisen, aber nicht deren Abwesenheit. Die formale Verifikation hingegen benutzt standardisierte Formate und kann die Abwesenheit von Programmfehlern nachweisen. Leider zeigen auch erfolgreiche Techniken dieser Art Schwächen. Kombinationen von mehreren Techniken versuchen, die Stärken sich ergänzender Techniken zu kombinieren, aber solche Kombinationen sind oft als zusammenhängende, monolithische Einheiten konzipiert. Sie sind unflexibel und es ist kostspielig, Techniken auszutauschen.

Um diesen Stand der Technik zu verbessern, ermöglichen wir eine Kooperation zwischen existierenden Verifikationswerkzeugen mit standardisierten Austauschformaten, ohne dass Anpassungen an den verwendeten Werkzeugen notwendig sind.

Zunächst arbeiten wir an einer Standardisierung der automatischen Testgenerierung für C. Wir erhöhen die Vergleichbarkeit von Testgeneratoren durch ein einheitliches Benchmarking-Framework und zuverlässige Werkzeuge, und stellen Instrumente zum Vergleich von Testgeneratoren und formalen Verifizierern zur Verfügung.

Als nächstes erstellen wir neue Konzepte für die Zusammenarbeit zwischen existierenden Verifizierern (sowohl Testgeneratoren, als auch formale Verifizierer). Wir demonstrieren die Flexibilität dieser Konzepte durch mehrere Kombinationen und durch die Anwendung auf das Problem der inkrementellen Verifikation. Weiterhin zeigen wir, wie bestehende, stark gekoppelte Techniken der Softwareverifikation in lose gekoppelte, eigenständige Komponenten zerlegt werden können, die durch klar definierte Schnittstellen und standardisierte Austauschformate kooperieren.

Alle präsentierten Konzepte werden von umfangreichen Implementierungen gestützt. Ausführliche experimentelle Evaluationen zeigen die Vorteile unserer Arbeit.

Durch unsere Arbeit verbessern wir die Vergleichbarkeit von automatisierten Verifikationswerkzeugen, ermöglichen Kooperationen zwischen vielen existierenden Verifikationswerkzeugen, erhöhen die Effektivität von Softwareverifikation, und schaffen neue Chancen für weitere Forschung im Bereich der kooperativen Verifikation.

Abstract

There are two major methods for software verification: testing and formal verification. To increase our confidence in software on a large scale, we require tools that apply these methods automatically and reliably. Testing with manually written tests is widespread, but for automatically generated tests for the C programming language there is no standardized format. This makes the use and comparison of automated test generators expensive. In addition, testing can never provide full confidence in software—it can show the presence of bugs, but not their absence. In contrast, formal verification uses established, standardized formats and can prove the absence of bugs. Unfortunately, even successful formal-verification techniques suffer from different weaknesses. Compositions of multiple techniques try to combine the strengths of complementing techniques, but such combinations are often designed as cohesive, monolithic units. This makes them inflexible and it is costly to replace components.

To improve on this state of the art, we work towards an off-the-shelf cooperation between verification tools through standardized exchange formats.

First, we work towards standardization of automated test generation for C. We increase the comparability of test generators through a common benchmarking framework and reliable tooling, and provide means to reliably compare the bug-finding capabilities of test generators and formal verifiers.

Second, we introduce new concepts for the off-the-shelf cooperation between verifiers (both test generators and formal verifiers). We show the flexibility of these concepts through an array of combinations and through an application to incremental verification. We also show how existing, strongly coupled techniques in software verification can be decomposed into stand-alone components that cooperate through clearly defined interfaces and standardized exchange formats.

All our work is backed by rigorous implementation of the proposed concepts and thorough experimental evaluations that demonstrate the benefits of our work. Through these means we are able to improve the comparability of automated verifiers, allow the cooperation between a large array of existing verifiers, increase the effectiveness of software verification, and create new opportunities for further research on cooperative verification.

Acknowledgements

I owe heartfelt thanks to a number of people that supported, motivated, and sometimes endured me.

First of all, my late mum. Mum, despite your hurricane-like personality you managed to push me towards computer science with prophetic peace. The occasional snarky comment about grades aside, you never expected anything but always believed. You put me on this path that I have been enjoying for so long.

Leonie, my wife and secret crush. Leonie, I strive for your wit and ambition. Your hand in mine lifts the pressure of a PhD into a breeze, and I look forward to all the things yet to come.

Dirk, my PhD supervisor. Dirk, you opened more doors than I can count. You always had time, always understood, always provided ideas and feedback. You infected me with your enthusiasm and motivated me to the sprints so important for academic publication. I could not have asked for a better mentor.

Off the academic track, I have to thank my dad. Dad, there is little chance you will ever understand my research topic. But I can always count on you, and your constant care and help is of immeasurable value.

My collaborators, colleagues, and student assistants—some of you filled more than one of these roles over the past years. In alphabetic order: Heike Wehrheim, Jan Haltermann, Lars Grunske, Marie-Christine Jakobs, Matthias Dangl, Matthias Kettl, and Michael Tautschnig; Gidon, Henrik, Karlheinz, Martin, Marvin, Nian-Ze, Nico, Philipp, Po-Chun, Stefan, Stephan, Sudeep, and Thomas; Klara Cimbalnik, Max Wiesholler, Valentin Port, and all the others. Thank you for the fun and productive work, the ideas, chatter, and things I learned from you. A special mention is due to Philipp. You supervised me for many years when I was a student assistant, and by now it is uncountable how often you immediately jumped to help when there were technical issues or I needed guidance.

Last, I want to thank Jan Strejček and Jonathan Bell: Thank you for giving your time and effort for reviewing this thesis.

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Contributions 2
1.3 Structure 3
1.4 Related Work 4

2 Background 5
2.1 Control-flow Automata as Program Representation 5
2.2 Automated Test Generation 6
2.3 Automated Formal Verification 9
2.4 Methodology 14

3 Towards Standardizing Test Generation (B.1–B.5) 17
3.1 A Level Playing Field for Test-Generation Comparison 17
3.2 Tooling for Comparison of Test Generators 20
3.3 From Verification Witnesses to Tests 21
3.4 The Current State of the Art in Bug-Finding 22

4 Towards Cooperative Software Verification (B.6–B.9) 23
4.1 Encoding Condition Automata 23
4.2 Cooperation between Test-Generators 24
4.3 Condition Automata for Difference Verification 26
4.4 Decomposing Verification Techniques 27

5 Future Research and Conclusion 29
5.1 Future Research 29
5.2 Conclusion 31

Bibliography 33

A Credits 45

B Original Manuscripts **47**
B.1 Software Verification: Testing vs. Model Checking 48
B.2 Plain Random Test Generation with PRTEST (Competition Contribution) 64
B.3 TESTCOV: Robust Test-Suite Execution and Coverage Measurement . . . 67
B.4 Tests from Witnesses: Execution-Based Validation of Verification Results 71
B.5 Six Years Later: Testing vs. Model Checking 92
B.6 Reducer-Based Construction of Conditional Verifiers 106
B.7 Conditional Testing: Off-the-Shelf Combination of Test-Case Generators . 118
B.8 Difference Verification with Conditions 138
B.9 Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR 160

List of Figures

2.1 A program as C code and as CFA 6
2.2 Example for coverage measurement 7
2.3 Test generation for program P and coverage criterion φ 8
2.4 Test-suite adequacy evaluation with a test oracle 8
2.5 The `metadata.xml` of a Test-Comp 2022 test suite 9
2.6 Shortened test XML of a Test-Comp 2022 test suite 9
2.7 Observer automaton for the safety property that reach_error() may never be called 10
2.8 Observer automaton for the safety property that program location l_4 may never be reached 10
2.9 Exemplary program abstraction for Fig. 2.10 10
2.10 Concrete state space of Fig. 2.1 10
2.11 Verification of program P with regards to property ϕ 11
2.12 Validation of violation witness $\mathcal{VW}$ 11
2.13 Conditional verifier 13
2.14 Difference in state-space restriction between violation-witness automaton and condition automaton 13
2.15 An sv-benchmarks task-definition file 15

3.1 Example of wrong branch-coverage report by GCOV 21

4.1 Reduction of original program and condition into residual program 24
4.2 Reducer 25
4.3 Test-Goal Extractor 25
4.4 Difference Computation 25
4.5 Component-based CEGAR 27

“Have you tried turning it off and on again?”

1 Introduction

1.1 Motivation

Our lifes have been infiltrated by software. Odds are there is a smartphone in your arm's reach or a small computer around your wrist; planes, cars and bicycles increasingly rely on software; the statistical methods of machine learning are used for personal entertainment, healthcare, research, and jurisdiction. Software supercharges humanity.

This reliance can have grave consequences when software malfunctions: (a) The heartbleed bug [132] in the cryptography library OpenSSL and the log4j bug[1] in the logging library log4j introduced severe security holes in everyday web services. The heartbleed bug was fixed two years after its introduction, and the log4j bug was mitigated only eight years after its introduction. (b) The Mariner 1 spacecraft (due to a faulty equation) [129], the Mars Climate Orbiter (due to wrong unit conversion) [14], and the Ariane 5 rocket (due to an integer overflow) [78] all three malfunctioned due to software bugs, leading to a combined loss of about US$ 600 million. (c) Statistical bugs in the brain-imaging software AFNI [8] question the results of about every tenth publication on brain-imaging research within a 15 year period, and a bug in radiation-therapy machine Therac-25 [105] killed three patients during treatment with lethal radiation overdoses. The "fixed" software introduced a new integer overflow that killed once more.

These examples are extremes, but, with lesser consequences, software does not behave as intended on a daily basis: software bugs are so common that we discard them as a law of nature: *software has bugs*. But it should not have to be this way.

To be able to fully trust software, many things have to come together. One piece of this bigger puzzle is the verification of *functional safety* of software: the check that software behaves as intended by their creators. Software systems are increasingly complex, and manual verification does not scale well. In turn, we need tools that automatically and reliably verify large-scale software for us. The holy grail we are working towards is a tool that fully automatically verifies any given software system for us at the push of a button, and tells us whether it is safe or not.

Because of its wide use in industry, we only consider tools for verification of C programs. We look at two categories of automatic software verification: Automated test generation (for software testing), and formal verification (which creates formal correctness proofs or counterexamples).

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44832

Approaches to automatically verify software [22, 26, 69] have been extensively researched since Alan Turing [3], but each existing verification technique has different strengths and weaknesses.

First, we need to understand these strengths and weaknesses. This requires the experimental evaluation of techniques on a level playing field. Tools for test generation in C have been living next to each other in isolation and had no standardized formats for testing or information exchange. But this is a fundamental necessity for fair evaluation (and later cooperation). We first work towards standardizing formats and improving the comparability of the different tools through reliable experimental evaluation.

Next, cooperation is necessary: two complementing techniques should not just work side by side, but cooperate, exchanging information to help each other solve tasks that neither of the techniques could solve on its own. Many research [19, 40, 89, 97, 107, 108, 109, 112, 115, 119] has combined verification techniques, but these approaches implement combinations as cohesive units, making them monolithic; inflexible and costly to exchange individual techniques.

We contrast this: We believe that cooperation must be possible through clearly defined interfaces and in a plug-and-play fashion. This avoids technology lock-in and enables quick adaption of new technologies. Such an off-the-shelf capability for cooperation requires formats for information exchange. So, as a second step, we inspect an existing, but hardly used exchange format for formal verification: condition automata [56]. We make it widely applicable, explore the opportunities it yields, and transfer the concept from formal verification to test generation.

Last, we show how existing, cohesive verification techniques can be decomposed into individual components with clearly defined interfaces, increasing their flexibility.

We hope that these building blocks contribute towards safer software.

1.2 Contributions

We provide the following contributions:

- We are the first to provide a common framework (Section B.1) for automated test generation in C, and we provide additional concepts and tooling for the comparison (Sections B.2 and B.3) and interoperability (Section B.4) of test generators and formal verifiers.
- We show the potential of formal-verification techniques for testing (Section B.1). Vice versa, we make formal verification more approachable to software developers through testing (Section B.4).
- We make the cooperation technique *conditional model checking* widely usable by using C programs as medium for information exchange (Section B.6), instead of the existing proprietary (hardly supported) format.
- We provide a conceptual framework for composition, cooperation and information exchange between test generators, by adapting the ideas of conditional model checking to testing (Section B.7).
- We show the flexibility of conditional model checking through the application to incremental formal verification (Section B.8).

- On the example of counterexample-guided abstraction refinement, we show that established, cohesive approaches to formal verification can be decomposed, to allow for more flexibility and fast adaption of new techniques (Section B.9).

All of our work is supported by rigorous implementations. Throughout our work, we developed the following tools:

- TBF (Section B.1) is an automatic test generation and execution framework for C programs. It is able to prepare C programs for multiple (back then) state-of-the-art test generators, create test harnesses for the generated tests from different proprietary formats, and execute the tests.
- PRTest (Section B.2) is a test generator for C programs. It produces test inputs with a uniform random distribution and performs incremental test-suite reduction based on branch coverage.
- TestCov (Section B.3) is a tool for test-suite execution and coverage measurement, with a focus on robust execution and reliable measurement.
- CondTest (Section B.7) is a framework for composition of test generators and verifiers through conditional testing [58].

We have also contributed multiple components to the software-verification framework CPAchecker [39] (Sections B.4, B.6, and B.8) and to the cooperative-verification framework CoVeriTeam (Section B.9).

Our research is published in top-tier conferences, including the IEEE/ACM International Conference on Automated Software Engineering (ASE) and the International Conference on Software Engineering (ICSE). Our article Software Verification: Testing vs. Model Checking (Section B.1) received the best paper award at the Haifa Verification Conference 2017 (HVC 2017).

1.3 Structure

This work connects a selection of my academic publications and is structured as follows: Chapter 1 explains the motivation, contributions, and related work of this thesis. Chapter 2 provides a broader context and necessary background. Two chapters then provide an overview of the selected publications: Chapter 3 is concerned with my work towards standardizing test generation, and Chapter 4 is concerned with my work towards cooperative verification and cooperative testing with the help of conditions. Last, Chapter 5 presents potential for future research and concludes this work. Appendix A states my contributions to each of the works presented in this thesis. Appendix B contains the original print versions of the presented works.

1.4 Related Work

Each of the works in Chapter B discusses its related work separately. Here, we give an overview on other research fields that also aim to ensure safe software. These can be put into three categories: (1) automatically generate a correct program, (2) automatically verify that a program is correct, and (3) interactively verify that a program is correct.

Correctness by Construction. Program synthesis [73] tries to automatically generate a program that fulfills a given specification. Many synthesis techniques are safe by construction—i.e., it is proven that a program synthesized through these techniques fulfills its specification. Examples are enumerative algorithms [73], deductive program synthesis [94], counterexample-guided inductive synthesis [82, 113], and synthesis based on constraint-solving [118]. Despite recent advances, program synthesis is mostly limited to generating individual functions, no full programs.

Automated Formal Verification. We aim to improve the performance of automated verification through cooperation. But research also continues on individual algorithms and techniques:

New proof techniques [1, 50, 96, 101] try to compute program proofs faster or find proofs for problems not solvable before.

New strategies for counterexample-guided abstraction refinement [2], improvements to static program slicing [84], and new abstract domains [12, 20, 74] try to scale formal verification through improved program abstractions.[2]

Compositional techniques [20, 37, 92, 109] analyze program procedures separately and scale well if fitting procedure summaries are found, but inferring these procedure summaries may be costly.

Distributed analyses [38, 83, 117] distribute the effort on multiple workers, effectively reducing the wall time of the analysis while the effort stays the same.

Interactive Formal Verification. In contrast to fully automated verification techniques, deductive verification [75] requires the user to provide contracts about the program, usually about the program procedures' pre- and post-conditions, as well as invariants of unbounded loops. These contracts are then checked and, if correct, used by a deductive-verification tool to check the correctness of the program.

Deductive verification scales well: Each program procedure can be analyzed independently, and, if a procedure is proven correct, the procedure's post-condition can be used as summary at each call-site. But it requires the user to provide and refine the procedure contracts to help the deductive verifier find a proof. These contracts can become lengthy for non-trivial procedures and may require many person-hours to write.

[2] We also published work in this area, introducing counterexample-guided abstraction refinement for symbolic execution [65].

2 Background

2.1 Control-flow Automata as Program Representation

Throughout our work, we use simplified views of programs. We assume imperative, sequential programs. All variables are unbound integer types from $\mathbb{Z}$, all program operations are defined over variables and integers. The set *Vars* contains all program variables. At program entry, variables can have any value. We use two classes of program operations: variable assignments and assumptions. A variable assignment $x = exp$ assigns the value of an expression *exp* to program variable $x \in \mathit{Vars}$. An assumption $[p]$ restricts the program's control flow to program states that fulfill the condition p. The set *Ops* contains all possible program operations. We generally describe our approaches for intraprocedural programs and do not consider function calls. But there are two function-call-like exceptions with a special meaning in our representation: (1) To signal the introduction of non-deterministic values in the program, we use assignments `x = nondet()`, `x = __VERIFIER_nondet_int()`, or the question mark `x = ?`. We call program variables that are assigned a non-deterministic value *input variables*. In practice, an input variable may receive its value from a sensor, a file, the program user, et cetera. (2) To signal a specification violation, some publications use statements `reach_error()`, `__VERIFIER_error()`, or similar. Throughout this work, we combine programs under analysis with information that is expressed as automata or graphs. Thus, a graph-based view of programs is helpful: We represent programs as control-flow automata (CFA) [57]. A CFA $P = (L, E, l_0)$ is a graph with nodes L (program locations), edges $E \subseteq L \times \mathit{Ops} \times L$ (control-flow edges), and an entry l_0 (program entry). A CFA represents the program's control flow, starting at program entry l_0. Transition from a program location l to another l' is possible if an edge (l, op, l') exists between the two and there is a valid evaluation of *op* according to the assumed program semantics. If a CFA node l has a predecessor node with more than one successor node, then l starts a new *branch*.

Figure 2.1 shows an example program, represented in the C programming language and as CFA. Here, variable `n` is an input variable. The program starts at l_1. Through evaluation of assignments `x = 0` and `n = nondet()`, control goes to program location l_3. Locations l_4 and l_7 start new branches: control goes to l_4 if assumption `[x < n]` is satisfiable. Otherwise, control goes to l_7.

We represent the program-state space as concrete states $\mathcal{C} \subseteq L \times (\mathit{Vars} \to \mathbb{Z})$, where the first element $l \in L$ represents the current location in the program.

```
int main(void) {
  unsigned int x = 0;
  unsigned short n = nondet();
  while (x < n) {
    x += 2;
  }
  if (x % 2 == 0) {}
  else
    reach_error();
}
```

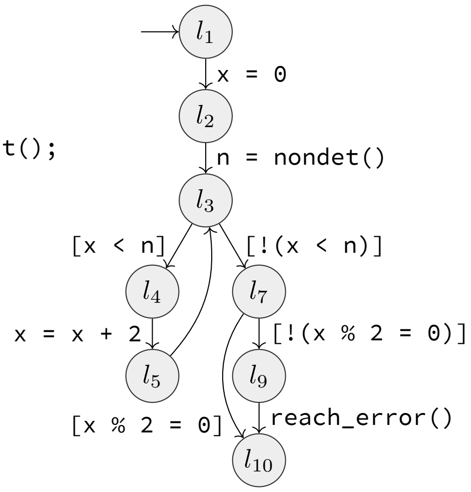


Figure 2.1: A program as C code and as CFA

2.2 Automated Test Generation

Software Tests. A test input $t = \langle v_0, \ldots, v_n \rangle$ is a sequence of n input values for a single program execution. When an input variable gets assigned a new non-deterministic value during program execution with t, the next input value v_i is used for the assignment. For example, given the program

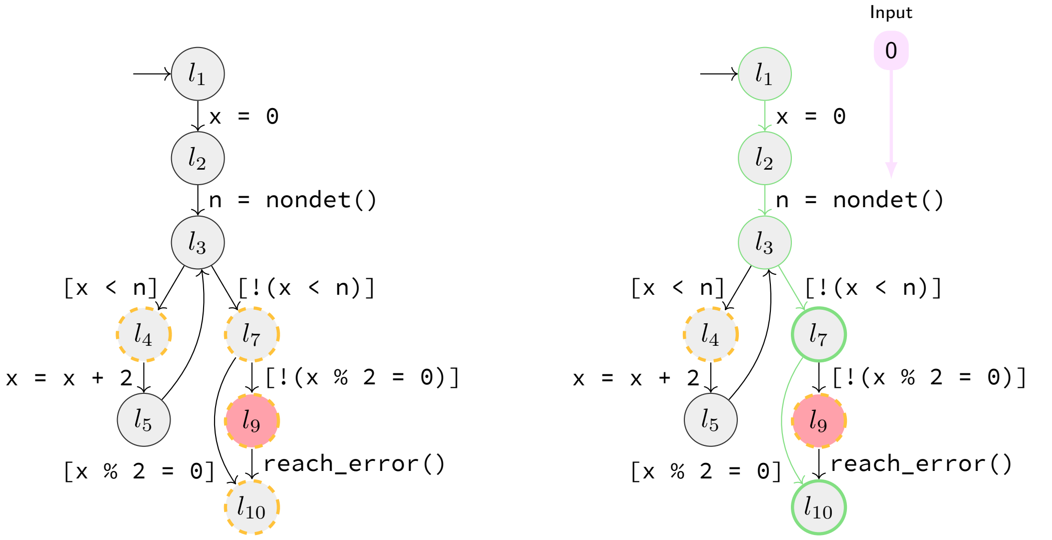
```
int a;
for (int i = 0; i < 2; i++) {
    a = nondet();
}
```

and test input $t = \langle 224, 65 \rangle$, the program execution with t first assigns 224 to `a`, and then assigns 65 to `a`. Given the program

```
int a = nondet();
int b = nondet();
```

and the same t, the program execution with t assigns 224 to `a` and 65 to `b`.

A software test consists of a test input and an expected program behavior. In our work, the expected program behavior is extrensic to the test: When we *generate a test*, only the test input is generated, no expected behavior for this input. The expected behavior is either available separately (e.g., function `reach_error` may never be called, there may be no unsigned overflow in the program), or implied (e.g., the program should never crash). From now on we use the term *test* interchangeably with the term *test input*. A collection of tests is called a *test suite*.



(a) Program with two sets of coverage goals: Coverage goals for branch coverage are l_4, l_7, l_9, l_{10}. Coverage goal for `reach_error()` is l_9.

(b) Test coverage recorded during execution of test with input value '0': $l_1, l_2, l_3, l_7, l_{10}$

Figure 2.2: Example for coverage measurement

Coverage Goals. It is impossible to proof program safety through testing for all but the simplest programs. Instead, the confidence in a test suite's expressiveness is indicated with *adequacy criteria*. Adequacy criteria can be based on different information: Examples are a behavior specification of the program (e.g., category partitions or state transitions), extrinsic information (e.g., seeded faults or program mutations [7, 103, 104]), or the program structure (e.g., error methods, branch coverage, or modified condition/decision coverage [88]). We focus on the latter. For adequacy criteria based on the program structure, the required and actual code coverage can be precisely measured—because of this, we call these *coverage criteria*. The specification language FQL [11] provides flexible means to formally define them. A coverage criterion consists of a set of *coverage goals*. A coverage goal is a projection on the CFA: a program location, edge, or any combination thereof [11]. To cover a coverage goal, a test must cover all components of the coverage goal in a single execution. For example, coverage goal @l_3 (written in FQL) requires a test's execution to pass through l_3, and coverage goal @l_3.@l_4.@l_4 requires a test's execution to first pass through l_3 and then at least two times through l_4. For the sake of presentation, we only consider single program locations (like @l_3) as coverage goals. Program transformations [98] can be used to map the coverage goals of a coverage criterion from one to another.

We first consider the coverage criterion that all calls to function `reach_error` are covered, defined in FQL as `COVER EDGES(@CALL(reach_error))`. For the program from Fig. 2.1, the only call to `reach_error` is at l_9. Let us assume we had a test suite $\mathcal{TS}_{\{0\}}$ that contains a single test with input value 0 for the single input variable `n`. Figure 2.2(b) shows in green all program locations that $\mathcal{TS}_{\{0\}}$ covers: l_1, l_2, l_3, l_7, and l_{10}. Since it

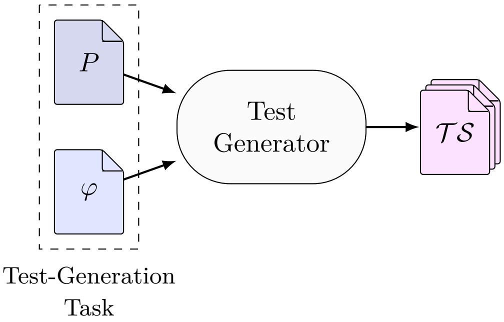


Figure 2.3: Test generation for program P and coverage criterion φ

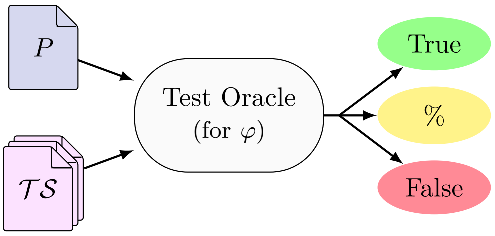


Figure 2.4: Test-suite adequacy evaluation with a test oracle

does not cover l_9, $\mathcal{TS}_{\{0\}}$ is not an adequate test suite. (Note that testing can not prove that l_9 is actually never reachable.)

Next, we consider the coverage criterion *branch coverage*, also known as decision coverage: It requires that all branches in the program are covered by the test suite, defined in FQL as `COVER EDGES(@DECISIONEDGE)`. In Fig. 2.2(a), the coverage goals for branch coverage are highlighted with a dashed, orange outline: $@l_4$, $@l_7$, $@l_9$, and $@l_{10}$. Test suite $\mathcal{TS}_{\{0\}}$ covers 2 out of 4 goals: $@l_7$ and $@l_{10}$. This is reported as 50 % coverage.

Test Generation. A test generator aims to generate a test suite for a given program and a given coverage criterion (Fig. 2.3). While a software developer that writes tests manually can be considered a test generator, we only consider automated generators that are not guided by any user feedback. Different approaches exist for test generation [18, 40, 67, 76, 84, 85, 87, 91, 95, 108, 115, 116, 124]. From purely random [124], over coverage-guided fuzzing [95, 114], to exhaustive state-space exploration [18, 40, 87]. We do not propose new generation strategies, but use, combine, and evaluate existing tools.

Test Oracle. A *test oracle* (Fig. 2.4) for a given adequacy criterion receives the program under test and a test suite and returns whether (or to which degree) this test suite fulfills the adequacy criterion on the program under test. Up to now (2022), all four editions of Test-Comp use our tool TestCov (Section B.3) as test oracle.

Test-Suite Representation. The First International Competition on Software Testing (Test-Comp 2019) [25] introduced a common format for test suites. In this format, a test suite is represented by multiple files: A `metadata.xml` (Fig. 2.5) provides information about the test suite, and for each test there is one individual file (Fig. 2.6) that defines this test's input values. All tools that participate in Test-Comp must output generated test suites in this format. This improves comparability and enables an easier integration of test generators in combination frameworks (Section B.7).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE test-metadata PUBLIC "+//IDN sosy-lab.org//DTD test-format
↪  test-metadata 1.0//EN"
↪  "https://sosy-lab.org/test-format/test-metadata-1.0.dtd">
<test-metadata>
  <entryfunction>main</entryfunction>
  <specification>COVER( init(main()), FQL(COVER EDGES(@DECISIONEDGE))
  ↪  )</specification>
  <sourcecodelang>C</sourcecodelang>
  <architecture>64bit</architecture>
  <creationtime>2021-12-16T07:12:02.605508</creationtime>
  <programhash>2f962093ad51cdfe116605a386faa8d78d826b9f</programhash>
  <producer>FuSeBMC v.4.1.14</producer>
  <programfile>[...]/sqlite/sqlite_merged_comb.i</programfile>
</test-metadata>
```

Figure 2.5: The `metadata.xml` of a Test-Comp 2022 test suite

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE testcase PUBLIC "+//IDN sosy-lab.org//DTD test-format testcase
↪  1.0//EN" "https://sosy-lab.org/test-format/testcase-1.0.dtd">
<testcase>
  <input type="int">15</input>
  <input type="char">0</input>
  <input type="char">7</input>
  [...]
</testcase>
```

Figure 2.6: Shortened test XML of a Test-Comp 2022 test suite. Typing values is optional.

2.3 Automated Formal Verification

Program Properties. Programs have liveness properties and safety properties [111]. A liveness property requires that a program can always reach a wanted state. A safety property requires that a program never reaches an unwanted state. In this work, we focus on verification techniques for safety properties. From now on, we use the terms *property* and *safety property* interchangeably.

We can express safety properties as finite automata, called *observer automata*. For a program $P = (L, E, l_0)$, an observer automaton $\mathcal{O}_P = (Q, 2^E, \delta, q_0, F)$ consists of a set Q of states, the alphabet 2^E of sets of CFA edges, transitions $\delta : Q \times 2^E \times Q$, an initial state q_0, and a set $F \subseteq Q$ of accepting states. Accepting states represent violated properties. Figure 2.7 shows the safety property that we focus on in most of our work: If `reach_error()` is called at any point during program execution, the property is violated.

Any (set of) coverage goals can be expressed as safety property. For example, for coverage goal $@l_4$, the corresponding safety property is that the program may never

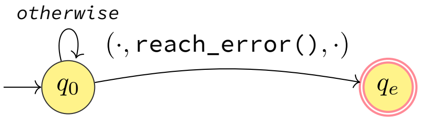


Figure 2.7: Observer automaton for the safety property that `reach_error()` may never be called

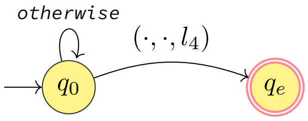
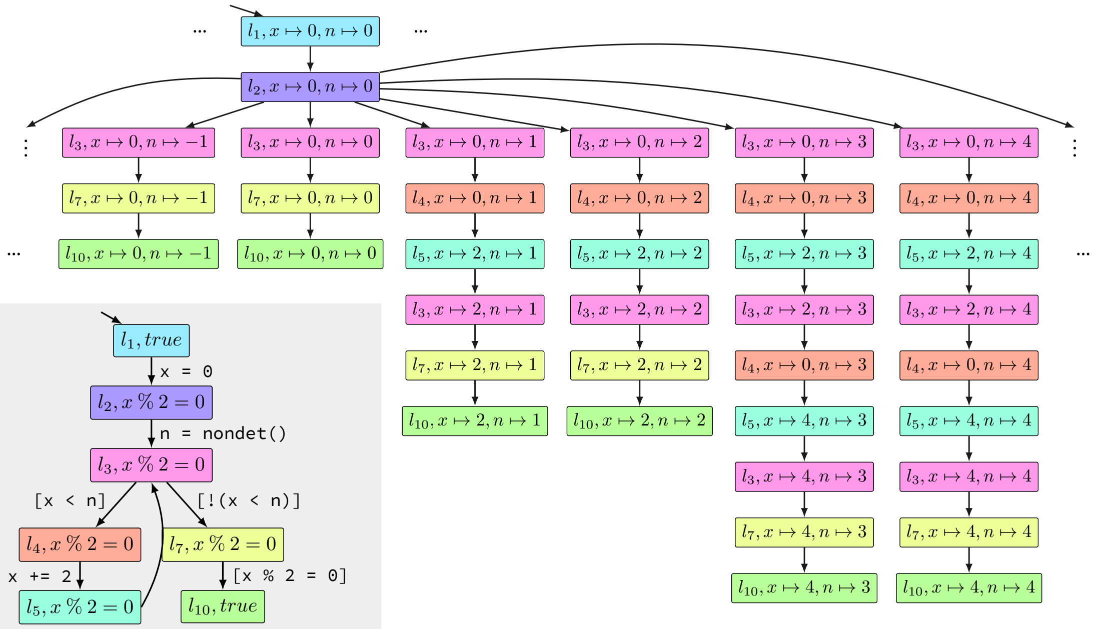


Figure 2.8: Observer automaton for the safety property that program location l_4 may never be reached


$l_1, x \mapsto 0, n \mapsto 0$
$l_2, x \mapsto 0, n \mapsto 0$
$l_3, x \mapsto 0, n \mapsto -1$
$l_7, x \mapsto 0, n \mapsto -1$
$l_{10}, x \mapsto 0, n \mapsto -1$
$l_3, x \mapsto 0, n \mapsto 0$
$l_7, x \mapsto 0, n \mapsto 0$
$l_{10}, x \mapsto 0, n \mapsto 0$
$l_3, x \mapsto 0, n \mapsto 1$
$l_4, x \mapsto 0, n \mapsto 1$
$l_5, x \mapsto 2, n \mapsto 1$
$l_3, x \mapsto 2, n \mapsto 1$
$l_7, x \mapsto 2, n \mapsto 1$
$l_{10}, x \mapsto 2, n \mapsto 1$
$l_3, x \mapsto 0, n \mapsto 2$
$l_4, x \mapsto 0, n \mapsto 2$
$l_5, x \mapsto 2, n \mapsto 2$
$l_3, x \mapsto 2, n \mapsto 2$
$l_7, x \mapsto 2, n \mapsto 2$
$l_{10}, x \mapsto 2, n \mapsto 2$
$l_3, x \mapsto 0, n \mapsto 3$
$l_4, x \mapsto 0, n \mapsto 3$
$l_5, x \mapsto 2, n \mapsto 3$
$l_3, x \mapsto 2, n \mapsto 3$
$l_4, x \mapsto 0, n \mapsto 3$
$l_5, x \mapsto 4, n \mapsto 3$
$l_3, x \mapsto 4, n \mapsto 3$
$l_7, x \mapsto 4, n \mapsto 3$
$l_{10}, x \mapsto 4, n \mapsto 3$
$l_3, x \mapsto 0, n \mapsto 4$
$l_4, x \mapsto 0, n \mapsto 4$
$l_5, x \mapsto 2, n \mapsto 4$
$l_3, x \mapsto 2, n \mapsto 4$
$l_4, x \mapsto 0, n \mapsto 4$
$l_5, x \mapsto 4, n \mapsto 4$
$l_3, x \mapsto 4, n \mapsto 4$
$l_7, x \mapsto 4, n \mapsto 4$
$l_{10}, x \mapsto 4, n \mapsto 4$

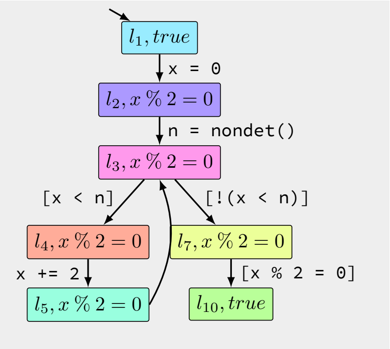


Figure 2.9: Exemplary program abstraction for Fig. 2.10

Figure 2.10: Concrete state space of Fig. 2.1

reach l_4 (Fig. 2.8). We exploit this relation in Section B.7 to make formal verifiers target branch coverage.

Program-State Reachability. To show that a program P violates a safety property represented by $\mathcal{O}_P$, verification techniques search for a counterexample to the property; i.e., a program execution whose sequence of CFA edges are an accepted input to $\mathcal{O}_P$. Such a sequence of CFA edges can also be implied by a test input. Analogous, to show that a program fulfills a safety property, verification techniques have to show that there exists no violating program execution. Figure 2.10 shows an excerpt of the infinite, concrete program-state space of Fig. 2.1. The concrete states are structured according to their predecessor-successor relation of a potential program execution. At the program entry l_1, there is an infinite number of possible concrete states for the still-unassigned program variables `x` and `n`. We restrict our excerpt to the program-state space for $x \mapsto 0, n \mapsto 0$.

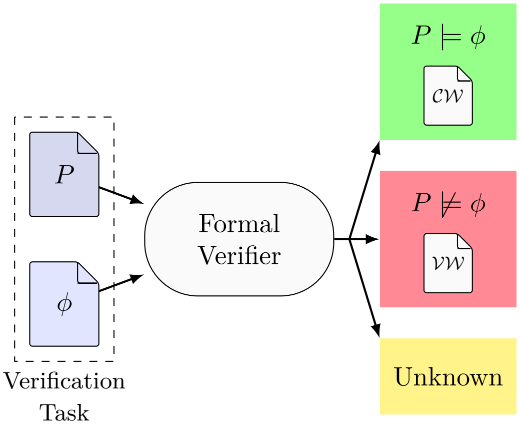


Figure 2.11: Verification of program P with regards to property ϕ

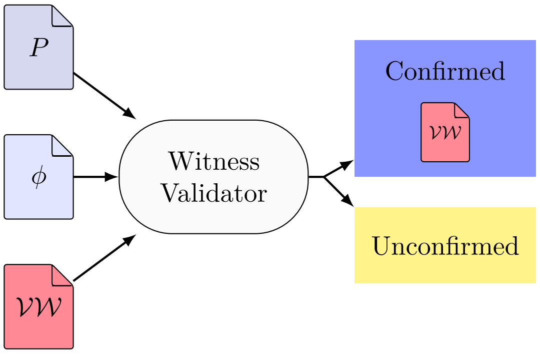


Figure 2.12: Validation of violation witness $\mathcal{VW}$ (analogous for correctness witnesses)

At l_3, because input variable `n` is not constrained, it may take any value. This means that the loop in the program may be unrolled an indefinite amount of times, and that there is an infinite number of possible concrete states.

Since testing can only prove a program safe through explicitly enumerating all feasible program executions like this, it does not scale.

Formal Verification. Formal verification [69] checks whether a program P fulfills a property ϕ, formally $P \models \phi$. If $P \models \phi$, we say that P is *correct with regards to* ϕ. If $P \not\models \phi$, we say that P is *incorrect with regards to* ϕ.

Most formal verification techniques [26] construct correctness proofs through abstraction [110]: they use an abstraction to the concrete program and exhaustively explore that abstraction's state space. If no state violates the program property, the program is safe with regards to the checked property. Figure 2.9 shows an abstract state-space exploration for Fig. 2.1 and the property that `reach_error` is never called. The exploration uses a simplified view on predicate abstraction [123] with predicates *true* and $x \,\%\, 2 = 0$. It represents sets of concrete states through the current program location $l \in L$ and either constraint *true* (any concrete state at l) or constraint $x \,\%\, 2 = 0$ (any concrete state at l whose variable assignment fulfills $x \,\%\, 2 = 0$). Formally, the concrete state space represented by such an abstract state is $[\![(l, p)]\!] = \{(l, \sigma) \in \mathcal{C} \mid \sigma \models p\}$.

The color of each abstract state in Fig. 2.9 signals the set of concrete states of Fig. 2.10 that it represents. Because the concrete value of `n` is arbitrary in all abstract states, and the value of `x` is known to be $x \,\%\, 2 = 0$ at l_7, the constructed abstract state space is finite: For each program location, it only requires a single abstract state that represents all concrete states possible at that location. Thanks to the predicate, the program abstraction can still proof that the CFA edge $(l_7, \texttt{[!(x \% 2 = 0)]}, l_9)$ that leads to `reach_error()` is never feasible.

Using a suitable program abstraction is essential for reliably proving a program safe. Different approaches exist for this (e.g., [12, 15, 21, 79, 122]) with different strengths and weaknesses. Chapter 4 works towards combining approaches to profit from their strengths and mitigate their weaknesses.

Verification-Result Witnesses. To increase the confidence in verification results, a formal verifier (Fig. 2.11) produces a verification-result witness: For $P \models \phi$ a correctness witness $\mathcal{CW}$, and for $P \not\models \phi$ a violation witness $\mathcal{VW}$. These witnesses can be checked by independent *witness validators* [49]. A witness validator (Fig. 2.12) tries to reconstruct the verification result from program P and property ϕ with the help of the witness. If this is successful, the verification result is *confirmed*. If it is not successful, the verification result stays *unconfirmed*. The verification result is not rejected per se, because the reason for a missing confirmation is not necessarily a wrong verdict; for example, validator CPA-w2t (Section B.4) can only validate violation witnesses that specify all input values. A witness validator can provide additional information to a confirmed verification result through a more detailed witness, called *testification* [48]. We use testification for violation witnesses in Section B.4 to generate from imprecise witnesses more precise witnesses that contain all information that is required to create test inputs.

A witness consists of two parts: metadata about the verification task, and a witness automaton with information that helps to recompute the claimed verification result. The type of witness automaton differs for violation witness and correctness witness. This work focuses on violation witnesses.

For a program $P = (L, E, l_0)$, a source-code guard $e \subseteq E$ is a set of CFA edges. A state-space guard $\psi \subseteq \mathcal{C}$ is a set of concrete program states. Set Φ contains all possible state-space guards. Violation-witness automata use source-code guards and state-space guards to restrict the program-state space to an (ideally small) subset that contains the claimed property violation.

For a program $P = (L, E, l_0)$, a violation-witness automaton $\mathcal{W} = (Q, \Sigma, \delta, q_0, F)$ is a finite automaton that consists of a set Q of states, alphabet $\Sigma \subseteq 2^E \times \Phi$ of source-code guards and state-space guards, transitions $\delta : Q \times \Sigma \times Q$, initial state $q_0 \in Q$, and accepting states $F \subseteq Q$ that represent that the claimed property violation is reached. For a transition $(q, (e, \psi), q')$, the source-code guard e allows the transition from q to q' if the next control-flow edge is in e. After transitioning to q', state-space guard ψ restricts the set of concrete program states to ψ.

In some of our work, we use violation witnesses with only source-code guards. In consequence, we omit state-space guards in these representations of violation-witness automata —at each edge, the state-space guard is the trivial guard $\mathcal{C}$.

For storage and exchange between tools, we use the exchange format[1] [49] for witnesses that is based on XML/GraphML [125]. This format is supported by all SV-COMP participants since SV-COMP 2015 [30] and is—to the best of our knowledge—the only

[1] https://github.com/sosy-lab/sv-witnesses

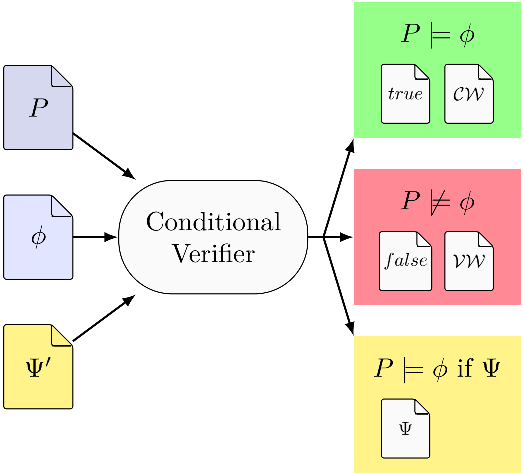


Figure 2.13: Conditional verifier

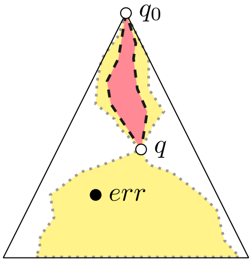


Figure 2.14: Difference in state-space restriction between violation-witness automaton (red, dashed) and condition automaton (yellow, dotted)

widely adopted exchange format for verification-results that allows to encode semantic information for reasoning about the result.

Conditional Verification. A verifier may neither be able to prove a property, nor find a counterexample. Reasons could be missing language support or resource exhaustion (e.g., the verification task requires more memory than available). In this case, a *conditional verifier* [56] (Fig. 2.13) produces a condition Ψ under which the program is safe. Another conditional verifier can then take Ψ as input (in addition to P and ϕ) to restrict its verification work to those parts of P that are not already covered by Ψ.

The first work on conditional verification [56] (on that we also base our work with formal verifiers [42, 44]) introduces conditions as finite automata: For a program $P = (L, l_0, E)$, a condition is a finite automaton $\Psi = (Q, \Sigma, \delta, q_0, F)$ of states Q, alphabet $\Sigma \subseteq 2^E \times \Phi$ of source-code guards and state-space guards, transitions $\delta : Q \times \Sigma \times Q$, initial state $q_0 \in Q$ and accepting states $F \subseteq Q$ that represent that the program is safe from this point forward. We say that any program execution leading to an accepting state is *covered* by Ψ. But if analysis reaches a state from which no accepting state is reachable, the program *may be* unsafe beyond this point. In practice, this is signaled with sink states (states without any outgoing edges).

Similar to violation-witness automata, a condition automaton restricts the program-state space to a sub-space of interest; but its interpretation is different: a violation-witness automaton restricts the program-state space to only those program paths that (it claims) end in a property violation; a condition automaton restricts the program-state space implicitly to program paths that it could not prove safe.

Figure 2.14 illustrates this. When an analysis reaches an accepting state $q \in F$ in a violation-witness automaton, it can always stop exploration of this sub-space (red, dashed area in Fig. 2.14) of the program-state space: either confirm a property violation at that state, or not. When a sink state q in a condition automaton is reached, this does not claim a property violation at that state. Instead, an analysis should verify the full state space that is still reachable from this location (yellow, dotted area below q in Fig. 2.14). This state space may contain a property violation (in Fig. 2.14, location err).

The similarity of violation-witness automaton and condition automaton is captured in the concept of protocol automata [48], a more generic type of non-deterministic finite automata that does not define the meaning of accepting states. This requires additional information about how to interpret the automaton. (for example, "like a violation-witness automaton" or "like a condition automaton").
Note: We silently generalized the original term *conditional model checking* [56] to the more generic term *conditional verification* [44] to signal that this concept is not limited to model checkers; it can be used with arbitrary types of verification techniques. *Conditional verification* is the conceptual idea described above, independent of the condition type and verifiers used. *Conditional model checking* is conditional verification with condition automata and formal verifiers.

CPAchecker. CPAchecker[2] [39] is a software-analysis framework with a modular structure. It consistently achieves good results at SV-COMP, is actively maintained, and is designed for integration and combination of new algorithms. CPAchecker supports witness generation as a verifier, violation-witness [48] and correctness-witness [47] validation, and conditional model checking [56]. For these reasons, we used it to implement multiple of our proposed concepts.

2.4 Methodology

Often, we show the effect of the proposed concepts and tools through experimental evaluations and comparison with the state of the art. For this, we follow the methodology of the International Competition on Software Verification (SV-COMP) [26] and the International Competition on Software Testing (Test-Comp) [22], explained below.

Benchmark Set. Throughout our work, we use the sv-benchmarks[3] benchmark set—the largest available benchmark set for automated software verification on C programs. In sv-benchmarks, a benchmark task can be a *verification task* or a *test-generation task*. A verification task [30] consists of a program (given as C source code) and a program property to check. The benchmark task also specifies the expected verification verdict. Multiple program properties exist in sv-benchmarks; our work focuses on the reachability property `unreach-call`, which specifies that a certain error method (`__VERIFIER_error` or `reach_error`) may never be called.

A test-generation task [32] consists of a program (given as C source code) and a coverage criterion that should be fulfilled by a generated test suite. Two coverage criteria exist: Criterion `coverage-error-call` requires a test suite to cover at least one call

[2] https://cpachecker.sosy-lab.org/

[3] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/

```
format_version: '2.0'

input_files: 'aws_add_size_checked_harness.i'

properties:
  - property_file: ../properties/unreach-call.prp
    expected_verdict: true
  - property_file: ../properties/coverage-branches.prp

options:
  language: C
  data_model: LP64
```

Figure 2.15: An sv-benchmarks task-definition file

to a certain error method (`__VERIFIER_error` or `reach_error`). Criterion `coverage-branches` requires a test suite to cover all program branches. For all benchmark tasks with coverage criterion `coverage-error-call` it is known that at least one error-method call is reachable in the program; i.e., the criterion can always be fulfilled. For benchmark tasks with coverage criterion `coverage-branches`, it is not known whether all program branches are reachable. This means that there may be tasks where coverage can not reach 100 %.

Tasks in sv-benchmarks are defined by task-definition files in the YAML format [32]. Figure 2.15 shows an example task-definition file that defines two tasks with program `'aws_add_size_checked_harness.i'`: One verification task with property `unreach-call` (which is true; i.e., no call to the error method is reachable), and one test-generation task with coverage criterion `coverage-branches`.

Benchmark tasks are grouped into categories. For example, category `ReachSafety-BitVectors` contains benchmark tasks that require a verifier to reason about bitwise operations (e.g., `x << 2` or `z = x ^ y`) and category `SoftwareSystems-AWS-C-Common-ReachSafety` contains benchmark tasks from the AWS C Common software library[4].

Each benchmark program uses methods `X __VERIFIER_nondet_X()` to introduce new non-deterministic values, where `X` is a primitive data type of C. These methods are declared, but not implemented. Their meaning is implied through the SV-COMP and Test-Comp rules. Example methods are `int __VERIFIER_nondet_int()` and `float __VERIFIER_nondet_float()`.

Computing Resources. In our experiments, we limit each tool execution on a benchmark task to the same limits as SV-COMP and Test-Comp: 900 s of CPU time and 15 GB of memory (RAM). Experiments are distributed on a cluster of equally configured machines with Intel processors—the latest iteration of the cluster uses 168 machines from 2015. Each machine runs Ubuntu Linux, has a single 8 core Intel Xeon E3-1230 v5

[4] `https://github.com/awslabs/aws-c-common`

CPU with 3.40 GHz, and 32 GB of memory. This computing power is similar to today's consumer machines.

Relevant Measurements. In our evaluations, we often use two measures to compare efficiency and effectiveness: CPU time and the overall number of results.

In contrast to the wall time, the CPU time is the process time the CPU actually spent computing. This measurement is agnostic of slow I/O operations and the number of CPU cores used by a tool. For example, if 4 CPU cores work 5 s each on a task in perfect parallelism, the wall time is 5 s, but the CPU time is 20 s ($4 * 5$ s). If a single CPU core waits 9 s for I/O and then computes for 1 s, the wall time is 10 s, but the CPU time is 1 s. Thus, the CPU time is useful to measure the efficiency of a tool: the actual computing effort the tool requires to perform a task.

For verification tasks, we differentiate between the following verification results:

- Correctly solved safe task (correct proof). The tool successfully claims the program to be correct with regards to the property.
- Correctly solved unsafe task (correct alarm). The tool successfully claims the program to be incorrect with regards to the property.
- Incorrectly solved safe task (incorrect proof). The tool claims the program to be correct with regards to the property, but it is actually incorrect.
- Incorrectly solved unsafe task (incorrect alarm). The tool claims the program to be incorrect with regards to the property, but it is actually correct.
- Unknown result. The tool did not reach a verdict.

The number of correct results reflects the effectiveness of a tool, while the number of incorrect results reflects its imprecision.

For test-generation tasks, we get the effectiveness of a tool through the achieved coverage of the generated test-suite (measured with TestCov, Section B.3).

Other measures for performance comparisons are wall time (already mentioned), memory usage, and energy consumption. We focus on run time to measure efficiency instead of memory usage, because, if our approaches fail to solve a verification task, this is almost exclusively because they reach the time limit. Energy consumption expresses the amount of energy (in kJ) the computation of a task requires. While this correlates with run time, there is no strict relation between the two. Thus, energy consumption would be a valuable supplement to run time, but reliable measurement has only become possible lately [23, 53]. To achieve reliable measurements of resource-consumption, we use BenchExec[5] [54].

[5] https://github.com/sosy-lab/benchexec/

3 Towards Standardizing Test Generation (B.1–B.5)

3.1 A Level Playing Field for Test-Generation Comparison

Software testing is universal in software development, but formal verification is hardly used. In the article "Software Verification: Testing vs. Model Checking" (Section B.1), we ignore potential other reasons for this, and examine whether testing is actually better at finding bugs than formal verification.

Formal verification is often seen as a means to prove the safety of software, which requires large effort and expertise. But our article shows that the program abstractions of formal verification techniques are effective at finding bugs, as well: In some way, a formal verifier that finds a fitting abstraction to a program is more precise than dynamic test execution, because actually irrelevant program parts are not explored. [70]

To show this, we compare the bug-finding capabilities of existing state-of-the-art test generators and formal verifiers for C programs. We select six test generators with different backgrounds: two test generators use formal-verification techniques tuned for testing [10, 34], one test generator uses symbolic execution [18], one test generator uses concolic execution [130], and one test generator uses random fuzz testing [95]. As a baseline, we implement a plain random tester (an early version of PRTest [124]). As formal verifiers, we select the four top-performers at finding bugs in the International Competition on Software Verification 2017 (SV-COMP 2017) [31].

Requirements for Reliable Comparison. For reliable comparison, it is necessary to have a well-defined benchmark-task set in a format that all tools understand. In addition, all tools have to use the same output format for verification results, so that results can be processed uniformly.

These conditions are met by formal verifiers: SV-COMP [24] establishes standardized formats [27] for verification tasks and verification results. For verification tasks, it uses the sv-benchmarks[1]. This benchmark set is community-driven and constantly expanding, contains explicitly annotated errors, puts great effort in removing all undefined behavior

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/

in programs[2], and structures benchmark tasks in categories with different focus. For verification results, SV-COMP uses the exchange format for verification-result witnesses [48].

For test generators for C, there existed no comparable standard before the first iteration of the International Competition on Software Testing [25]. To mitigate this, we built the framework TBF [64] that adjusts inputs to each test generator's expected format, and parses test generators' different output formats into a single unified structure.

Benchmark Set for Test Generation. The research in testing C programs has no standard benchmark to compare against: Many works [18, 77, 90, 91, 106] use different real-world C code, most prominently the GNU coreutils[3] and GNU grep[4]. But this real-world software contains undefined behavior that may be interpreted differently by different test generators and it is unknown how many bugs exist at which locations. To provide a well-defined alternative, we turn the verification tasks of sv-benchmarks into test-generation tasks.

Each of the selected test generators requires the program to mark inputs for which to generate test input in different ways:

- AFL-fuzz [95] requires a program under test to read test input from a file in any way it likes. The file is passed to the program as command-line argument.
- CPATiger [34] uses function `input()` to mark input; for example `x = input()`.
- Crest-PPC [130] uses special C macros `CREST_X(x)` with primitive type `X` to mark program variable `x` of type `X` as input; for example `CREST_int(x)`.
- FShell [9] assumes that any function without existing definition and any of C's scanf methods read input.
- Klee [18] uses the function `klee_make_symbolic(”x”, &x, sizeof(x))` to mark program variable `x` as input.
- PRTest [124] uses the sv-benchmarks methods. For example, the program statement `x = __VERIFIER_nondet_int()` introduces a new input value of type int.

Of the selected test generators, only FShell and CPATiger allow to specify a coverage criterion. We specify branch coverage. AFL-fuzz uses a coverage criterion similar to condition coverage, Klee and Crest-PPC use condition coverage. PRTest does not consider any coverage criterion, but indefinitely generates new random tests.

Test-Suite Formats. JUnit is a vastly popular framework for writing program tests for Java and often used in test-generation research [16, 72, 127]. But no comparably established framework exists for C. Instead, test generators for C produce test-suites in almost arbitrary formats. The test generators that we consider use the following formats:

- AFL-fuzz [95] stores each generated test as an individual file. The (binary) content of the file is the test input.
- CPATiger [34] stores all generated tests in a single file in a JSON-like structure.

[2] SV-COMP participants are incentivized to find undefined behavior in programs to improve their tool's performance in the competition.

[3] https://www.gnu.org/software/coreutils/

[4] https://www.gnu.org/software/grep/

- Crest-PPC [130] stores all generated tests in a single file. Each line in that file is an individual test with comma-separated input values.
- FShell [9] stores all generated tests in a single file; in a format similar to CPATiger, but not equal.
- Klee [18] stores each generated test as an individual file, in a proprietary binary format. It ships a tool for both inspecting and executing the generated tests. This tool outputs a list of the name, size and value of each input.
- PRTest [124] stores each generated test as an individual file. Each line in that file is one input value.

TBF. Our tool TBF adds implementations for all methods `__VERIFIER_nondet_X()` to match the test generator's expectations. For example, for Crest-PPC, TBF defines:

```
int __VERIFIER_nondet_int() {
  int __sym;
  CREST_int(__sym);
  return __sym;
}
```

TBF understands the six different output formats for test suites, converts them into a uniform internal structure, and executes that the same way for each tester.

For execution, TBF defines method `__VERIFIER_error` in the program under test to make its calls easily observable:

```
int __VERIFIER_error() {
  fprintf(stderr, ” __TBF_error_found.\n”);
  exit(1);
}
```

When `__VERIFIER_error` is called during a test execution, TBF reports that the test suite successfully covered the error. This way, TBF enables us to benchmark the bug-finding capabilities of test generators for C with the sv-benchmarks.

Results. We compare the results of the test generators executed with TBF to the formal verifiers of SV-COMP. We use the same machines and resource limits as SV-COMP 2017: 900 s of CPU time and 15 GB of memory.

To make sure that formal verifiers do not get an advantage because they are imprecise and guess lucky, we make sure that they are of high precision (only 3 false alarm across all 4 203 error-free verification tasks) and we run witness validation to confirm the reported bugs. The results show that the union of all formal verifiers is able to find bugs in more benchmark tasks than the union of all test generators: All formal verifiers find bugs in 979 tasks (with confirmed results), while test-generators only find bugs in 887 tasks. The best individual formal verifier (based on confirmed results) is CPA-Seq [100]: it finds bugs in 857 tasks. The best test generator is Klee [18]: it finds bugs in 826 tasks.

The union of all tools is also better than the union of all formal verifiers, with 1 068 tasks. This shows us that combinations between testing and formal verification may be fruitful.

Test-Comp. In 2018, Test-Comp introduced standardized formats for both benchmark tasks and test suites. Thanks to this, we do not require TBF anymore, but can perform comparative evaluations within the Test-Comp framework. In consequence, TBF's tool development stopped and the repository has been archived.

3.2 Tooling for Comparison of Test Generators

Test-Generation Baseline. Verifiers are often compared to the state of the art, but no comparison to a simple-to-understand baseline is made. This increases the risk that some easy-to-get coverage goals or program-language features are missed by all state-of-the-art approaches. In the article "Plain Random Test Generation with PRTest (Competition Contribution)" (Section B.2), we present the tool PRTest, a plain, random test generator for C programs. PRTest compiles a test harness that generates uniformly distributed test inputs against the program under test and executes it repeatedly. Because programs are compiled with common C compilers and no program characteristics are considered, PRTest supports the same language constructs that the compiler supports. This full language support and simple approach makes PRTest a good baseline for comparison. It participated in all iterations of Test-Comp and continuously shows that even the well-performing test-generation approaches have weaknesses.

For example, it is visible that in Test-Comp 2022 category *Cover-Error*, only the tools FuSeBMC [86], VeriFuzz [114], and LibKluzzer [76][5] can find bugs in all the tasks that PRTest can find the bugs for.

Reliable Coverage Measurement. In practice, to measure the code coverage of a test suite for C, gcov[6] and llvm-cov[7] are used. Unfortunately, these tools do not measure branch coverage based on the input source code, but based on an internal representation that splits Boolean operators into multiple branches. Figure 3.1(a) shows an example for this: Given the inputs -1 (for x) and 0 (for y), the program will enter the else-branch and the expected branch coverage is 50 %. But gcov and llvm-cov both report a coverage of 25 %, because the condition in line 6 is split into two branches internally (Fig. 3.1(b)).

In addition, there are known bugs [131] related to the tools' coverage measurement. Last, coverage measurement is accumulated across test executions. But test executions may influence each other (for example, because of file-system changes), and tests may provoke unwanted program behavior (e.g., spawning an indefinite amount of processes, or consuming all system memory).

[5] LibKluzzer fails to solve one task that PRTest can solve due to an out-of-memory error

[6] https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html

[7] https://www.llvm.org/docs/CommandGuide/llvm-cov.html

```
1  #include <stdio.h>
2  int main() {
3      int x; scanf(”%d”, &x);
4      int y; scanf(”%d”, &y);
5
6      if (x > 0 && y > x) {
7          return 0;
8      } else {
9          return 1;
10     }
11 }
```

(a) Program with two conditions for single branch (in line 6)

Input: -1, 0

```
   1*:    6:   if (x > 0 && y > x) {
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
branch  2 never executed
branch  3 never executed
#####:    7:        return 0;
    -:    8:    } else {
    1:    9:        return 1;
    -:   10:    }
    -:   11:}
```

(b) Branch coverage reported by gcov: 25 %. The expected branch coverage is 50 %.

Figure 3.1: Example of wrong branch-coverage report by gcov

We solve this issue in article “TestCov: Robust Test-Suite Execution and Coverage Measurement” (Section B.3). TestCov is a continuation of TBF’s internal test-suite execution. It provides robust coverage measurement based on the source code (e.g., for branch coverage) and it supports the XML-based exchange format for test-suites that Test-Comp established.

To execute tests reliably, it uses components of BenchExec [54] to isolate individual tests from each other (both file-system changes and resource usage), and uses program instrumentation to explicitly label coverage goals in the source code of the program under test. It can then use gcov’s reliable statement-coverage measurement to measure the code coverage of these inserted labels. This way, TestCov can measure branch coverage and error coverage of individual test execution and across a full test suite. In addition, it provides individual measurements of run time and memory consumption per test.

TestCov is under active development and has been used in all iterations of Test-Comp to date, to measure the coverage of test suites generated by all participants. In Test-Comp 2022 [22], TestCov successfully executed over 50 832 distinct test suites from 12 test generators, over 4 236 different test-generation tasks.

3.3 From Verification Witnesses to Tests

We have established comparability of test generators for C and shown that formal verifiers are well suited for finding bugs in software. Next, we turn formal verifiers into actual test generators, so that this potential can be used.

This has one additional advantage: Software developers are well versed in the use of tests and debugging tools, but often have little knowledge in formal verification. So to increase the usability of formal verifiers for bug-finding, previous work [33] on Blast proposed to turn the abstract counterexamples produced by formal verifiers into executable tests. Our article “Tests from Witnesses: Execution-Based Validation of Verification Results" (Section B.4) follows this idea and presents a method to convert the (abstract) violation witnesses that every formal verifier of SV-COMP supports into executable tests.

An executable test for a found fault gives the highest possible confidence in the reported alarm (because it can be directly observed through the execution), and it makes subsequent work with the alarm easy, because the developer can use existing tools for debugging. In addition, this technique allows to use any formal verifier for directed test generation (generating a single test for a violated property), given that they support the exchange format for witnesses—which all participants of SV-COMP do.

The article introduces two new witness validators, CPA-w2t and FShell-w2t. Both use the article's concept: They turn, if possible, a given violation witness into an executable test and execute it. If the test execution confirms the alarm, the violation-witness is accepted. Both CPA-w2t and FShell-w2t participated in SV-COMP since SV-COMP 2018.

3.4 The Current State of the Art in Bug-Finding

With the introduced approaches and the standardization through Test-Comp, we provide a large-scale comparison of tools for formal verification and automated test generation with regards to their bug-finding capabilities. Our article "Six Years Later: Testing vs. Model Checking" (Section B.5) compares all SV-COMP 2023 and Test-Comp 2023 participants based on the competitions' open data [28, 29]. This gives highest confidence on the reported results: tool developers can configure the tools for the competition and both the tools and the data is peer reviewed.

Compared to our previous study [64], we do not bother to report bug reports from formal verification that are not confirmed by at least one validator in SV-COMP. In addition, we report how many of the bug reports from formal verification can be confirmed through execution-based validation. Last, we check whether the test generators that participate in Test-Comp are tuned towards the coverage goal `coverage-error-call`, and how good the test suites generated for `coverage-branches` are in finding bugs.

Our study shows the following: (1) Many automated test generators have adapted hybrid approaches to bug finding, using multiple formal techniques or mixing formal and dynamic analysis techniques. The winner of Test-Comp 2023, FuSeBMC [86], combines input fuzzing [93] with bounded model checking [13]. (2) Automated test generators have greatly improved in the past years and surpass SV-COMP participants; but formal verification is still competitive. (3) Execution-based validation of verification results works well for a high number of results. This gives bug reports of formal verification the same level of confidence as bugs revealed through test execution. (4) Regarding targeting error calls, our results are not surprising, but give confidence: Almost all Test-Comp participants target the looked-for error call in their test generation.

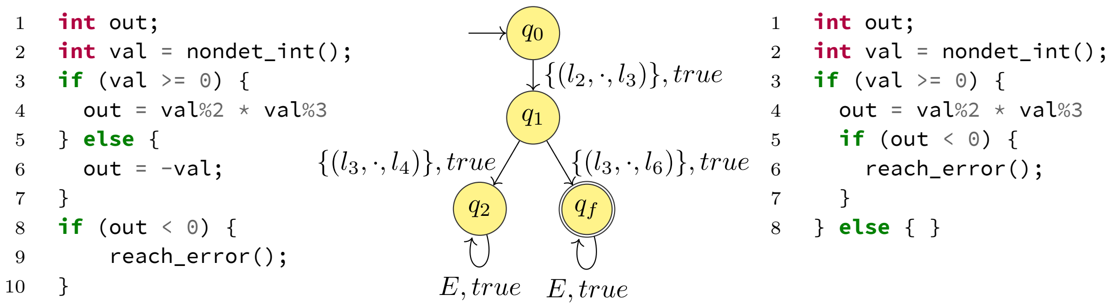
4 Towards Cooperative Software Verification (B.6–B.9)

4.1 Encoding Condition Automata

The idea of conditional verification [56] not only proposed condition automata, but also an exchange format for these. To the best of our knowledge, only CPAchecker supports this format to this date. But for successful cooperation, we need more than a single verifier. Instead of looking into reasons why the format for condition automata is not adapted by other verifiers, and instead of proposing a new format for conditional verification, we decided to use an existing format that every verifier understands: program source code.

Condition to Code. In the article "Reducer-Based Construction of Conditional Verifiers" (Section B.6), we define the notion of a *reducer*: A reducer takes a program P and a condition Ψ for P and creates a new residual program P_r that only contains those program executions that are not already covered by Ψ. Figure 4.1 shows an example for this: The program in Fig. 4.1(a) and the condition in Fig. 4.1(b) can be combined into a new residual program (Fig. 4.1(c)). This residual program does not contain any of the original program executions that go through the else-branch (line 6 in Fig. 4.1(a)).

Combining Formal Verifiers. By encoding the information of the condition in a new program, we can use any off-the-shelf verifier instead of a specialized conditional verifier. We perform a large experimental evaluation with three formal verifiers: CPAchecker, Smack and UAutomizer. Each of the three verifiers A is compared with a combination of CPAchecker's predicate abstraction and the reducer-based construction applied to A. The results are encouraging: we not only prove the feasibility of the reducer-based construction of conditional verifiers, but we also show the potential benefits of now-possible tool combinations: Each of the tool combinations can solve significantly more tasks than the respective stand-alone tool, and the overall effectiveness increases: Thanks to the tool combinations, we can now solve verification tasks that none of the stand-alone tools could solve before.



```
int out;
int val = nondet_int();
if (val >= 0) {
  out = val%2 * val%3
} else {
  out = -val;
}
if (out < 0) {
    reach_error();
}
```

q_0
$\{(l_2, \cdot, l_3)\}, true$
q_1
$\{(l_3, \cdot, l_4)\}, true$
$\{(l_3, \cdot, l_6)\}, true$
q_2
q_f
$E, true$
$E, true$

```
int out;
int val = nondet_int();
if (val >= 0) {
  out = val%2 * val%3
  if (out < 0) {
    reach_error();
  }
} else { }
```

(a) Program (b) Condition (c) Residual program

Figure 4.1: Reduction of original program and condition into residual program

Combining Formal Verifier and Test Generator. Last, we show an additional benefit of encoding the condition in the program: the concept of conditional verification, previously only used as conditional model checking with formal verifiers, is now applicable to arbitrary tools that analyze software. As example, we use test generators: We consider three test generators, AFL-FUZZ, CREST-PPC, and KLEE. We combine CPACHECKER's predicate abstraction with these through our reducer-based construction, and show that the test generators' effectiveness in finding errors on the residual programs increases compared to the original programs.

Later Developments. The introduction of residual programs enables an array of new applications: Subsequent work [41] explores different detail levels of residual programs. Difference verification (Section B.8), an application of condition automata, is usable with any off-the-shelf verifier thanks to residual programs. METAVAL [45] transfers the idea of encoding information as source code to violation-witness automata and correctness-witness automata, and turns any given off-the-shelf verifier into a witness validator. And component-based CEGAR (Section B.9) uses METAVAL to turn any off-the-shelf verifier into one of its components.

4.2 Cooperation between Test-Generators

Through the reducer-based construction of conditional verifiers, we can use test generators as a second component in conditional verification to solve a verification task. But we can not yet combine two test generators to solve a test-generation task.

Conditional Testing. In the article "Conditional Testing: Off-the-Shelf Combination of Test-Case Generators" (Section B.7), we transfer the idea of both conditional verification and residual programs to automated test generation: If a test generator is not able to create a test suite that fulfills a coverage measure to 100 %, we compute the remaining coverage goals and represent them as a condition. A conditional test generator

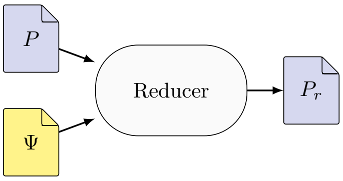


Figure 4.2: Reducer

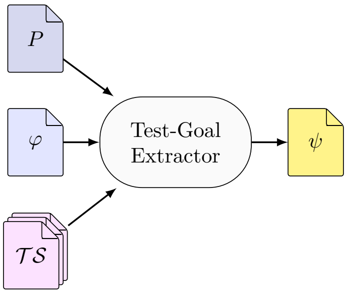


Figure 4.3: Test-Goal Extractor

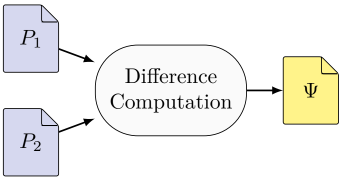


Figure 4.4: Difference Computation

can then take this condition and only generate tests for the remaining coverage goals. As condition, we use a type tailored to test generation: We express conditions as the set of remaining coverage goals. Because the notion of a *conditional test generator* is newly proposed, there are no supporting tools yet. To remedy this, we define program reduction and condition generation for conditional testing:

Program Reduction. We define the notion of program reducer in the context of conditional testing. A reducer (Fig. 4.2) for coverage goals is a testability transformation [98] that takes a program P and a condition Ψ (in the form of a set of coverage goals), and returns a residual program P_r. We require a program reducer to be *sound* and *complete* to be able to fulfill the original test-generation task across combinations. Both attributes are defined in Section B.7.

We show three exemplary types of reducers: the identity (the program stays unchanged), a program reducer that uses syntactic pruning of exhaustively covered program parts, and a program reducer that annotates relevant test goals in the program, for example to prove their (in-)feasibility with formal verifiers.

Condition Generation. We define the notion of a test-goal extractor: A test-goal extractor (Fig. 4.3) takes a program P, a coverage criterion φ and a test suite $\mathcal{TS}$, and returns the set ψ of covered test goals of φ. The information of a *sound* and *complete* test goal extractor (again, both attributes are defined in Section B.7) can then be used to generate a condition by computing the set of all test goals described by φ, and removing all already covered test goals from that set. At first sight it may seem like the construction of a condition would be easier if the test-goal extractor provided the number of currently uncovered test goals directly. But, (a) this would not fit the existing definition of conditions, which are required to encode the covered verification space, and (b) this would require the test-goal extractor to always know about all covered test goals, either through re-execution of all generated tests or by being stateful.

We instantiate a test-goal extractor based on program instrumentation, test execution and line-coverage measurement.

Combinations. To get access to a large selection of tools, we use the formats that are defined by Test-Comp and supported by all Test-Comp participants. With program reducer and test-goal extractor, this gives access to a large selection of off-the-shelf test generators as conditional testers.

The introduction of conditional testing allows different flavors of combinations between test generators; we propose a few conceptually (e.g., sequential, cyclic, parallel), and then go on to show the potential benefit of conditional testing through experiments on sequential combinations of pairs of test generators.

Turning Formal Verifiers into Test Generators. In addition, we show how to turn any formal verifier into a test generator through the witness-to-test generation [52] and a cyclic conditional-testing composition that restarts the formal verifier and the witness-to-test generation until all test goals are covered or proven infeasible.

Later Developments. We implemented conditional testing in the tool CondTest. Initially, CondTest was a stand-alone tool for composition of different test generators and formal verifiers for test generation. Since version 3.0, the composition moved to CoVeriTeam [55], and CondTest provides the reducers and test-goal extractor.

4.3 Condition Automata for Difference Verification

Condition automata specify the program-state space a verifier must prove safe. Originally, conditional model checking was proposed to make formal verifiers cooperate, but the control-mechanism of condition automata can also be used for other purposes: In the article "Difference Verification with Conditions" (Section B.8), we show the versatility of condition automata by using them for difference verification.

Assume that we have two revisions of a program, rev. 1 and rev. 2. We trust rev. 1, for example because it was fully verified or has been running in production for a long time without issues. Now, we want to verify rev. 2. Since we already trust rev. 1, we do not need to fully re-verify rev. 2—instead, it is enough to only verify those parts that might have changed from rev. 1.

We introduce an algorithm, DiffCond, that takes two revisions of a software code and creates a condition automaton that encodes the program-state space that is unaffected by the changes from one revision to the other (Fig. 4.4). This condition can then be given to a conditional verifier to only verify the parts of the program that may be affected by the change.

We show the benefits of this approach on incremental verification on a large number of benchmark tasks that we have generated from existing, strongly coupled programs. These new benchmark tasks were adopted by the SV-COMP iterations following this article, in the new category *ReachSafety-Combinations*.

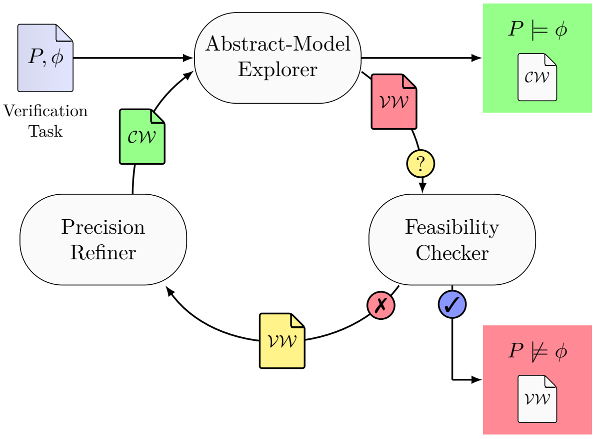


Figure 4.5: Component-based CEGAR

4.4 Decomposing Verification Techniques

The idea of multiple components working together towards a single verification task is visible in multiple traditional verification techniques, like CEGAR [68], k-induction [4], and static program slicing [99, 120]. To increase flexibility, ease maintenance and make it easier to see conceptual similarities [50], we propose to decompose existing techniques. We do this on the example of CEGAR [68].

CEGAR. CEGAR consists of three steps: (1) An abstract model of the program-under-verification is explored; the aim is to prove it correct with regards to a property ϕ, or find a counterexample to ϕ. Usually, at the beginning of a verification run, this abstract model is very coarse. If the abstract model is correct, the verification task is solved $P \models \phi$. (2) If a counterexample is found, it is checked for feasibility with a high-precision technique. If the counterexample is confirmed feasible, the verification task is solved $P \not\models \phi$. (3) If the counterexample is infeasible, the abstract model is made more precise so that this spurious counterexample is not encountered again in subsequent abstract-model explorations. After this precision refinement, the cycle repeats at (1) with the new, more precise model. These three steps are traditionally implemented as a single algorithm.

Decomposition of CEGAR. In the article "Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR" (Section B.9), we show that the three steps of CEGAR can be defined as individual software components that exchange information through verification artifacts. We call this concept *component-based CEGAR* (Fig. 4.5).

For information exchange, we show how to use the exchange formats for verification witnesses. This enables us to use existing tools in any of the three steps of component-based CEGAR: Correctness-witness validators as abstract-model explorer, violation-witness validators as feasibility checker, and violation-witness validators that produce correctness witnesses as precision refiner.

We implement a selection of different CEGAR compositions in CoVeriTeam [55], and show through an experimental evaluation that the technical disadvantages of this decomposition are manageable: there is only a constant factor of run-time decrease compared to the natively implemented CEGAR in CPAchecker.

Component-based CEGAR enables developers to exchange CEGAR components in a plug-and-play fashion. We hope that this not only avoids the lock-in effect, but also accelerates future research in CEGAR and serves as a positive example of decomposing existing verification techniques.

5 Future Research and Conclusion

5.1 Future Research

Mutation Testing. So far, we only considered traditional coverage measures as test-adequacy criteria. We do consider the bug-finding capabilities of test generators when we ask them to generate a test suite that covers a specific test (coverage criterion *coverage-error-call*). But mutation testing [7, 80, 103] goes beyond that: it not only evaluates whether a test suite covers a current bug in the program, but also whether future bugs would be detected. To include mutation testing in test-generation evaluation, TestCov could be extended to consider the mutation testing criterion for either *weakly killed* [128] or *strongly killed* [7] mutants. Existing mutation-testing tools [5, 71, 121] can be used for mutant creation.

Exchange Formats for Formal Verification. In this work we considered three exchange formats for formal verification: condition automata, violation witnesses, and correctness witnesses. All three are based on finite-state automata and describe a subspace of the program-state space. Unifying the three into a single exchange format may be useful; this would require verifiers to only implement a single format instead of three, and it would lessen the restrictions on input/output formats, which increases the interoperability between verifiers.

In addition, condition automata should be examined in more detail: CPAchecker writes a condition automaton as a precise projection of the unfinished parts of the explored abstract program-state space, based on the currently computed state space. These computed state spaces can be very large. In consequence, condition automata can grow very big (millions of lines of text description). Existing work [41] approaches the issue by reducing the size of a created residual program, but similar techniques may be used one step earlier, when writing the original condition automaton. The effect of different state-space traversal strategies should also be considered, to keep the unfinished parts of the explored abstract program-state space small; the spectrum goes from depth-first search (which keeps the description of the unfinished parts as small as possible because only a single abstract program path is explored at a time), to breadth-first search (which creates the maximum number of unfinished parts as all possible abstract program paths are considered concurrently).

Residual Programs. Currently, we create residual programs for formal verification as a product of program and condition automaton. This is precise, but may create large residual programs because every transition path in the condition automaton is directly encoded in the program. To create smaller residual programs, other techniques for program reduction could be considered, for example based on static program slicing [99], as inspired by other work on conditional model checking [102].

For test generation, we create residual programs through syntactic pruning of fully covered program sub-trees. More elaborate testability transformations [98] could be considered to create smaller residual programs.

Combinations. To keep the design of our experimental evaluations manageable, we only considered sequential combinations of verifiers. This means that many potential combinations are yet unexplored. Notable mentions are cyclic combinations of sequential tools, which have proven successful [40] for test generation in cohesive implementations, as well as the decomposition of a verification- or test-generation-task into a set of smaller tasks. This has been proven successful for formal verification [20, 92] and test generation [107]. Current techniques for formal verification provide no exchange format for decomposition information, and do not allow the use of different verifiers per sub-task. Techniques for test generation do decompose the input program, but have no means to adjust the coverage criterion accordingly.

Decomposed sub-tasks could also be combined with strategy selection [6, 17, 46, 126] to automatically choose a fitting verifier and configuration for each sub-task.

Information Use. We have only considered cooperation between verification techniques that work towards the same goal; solving a verification task, or solving a test-generation task. We discarded all sub-parts already solved (for example sub-trees of the program-state space that were fully explored or test goals). Symbiotic [96] also combines different verification tools, but uses the information differently: Results of initial (usually fast) analyses are used to simplify the verification task for a final, precise analysis. Analyses can also encode information in condition automata as state-space guards. This information could be used to achieve the same.

In addition, we do not check the soundness of a condition, but always assume it is correct. This means that we may introduce irrecoverable imprecisions if an imprecise analysis is used. A mechanism to check imprecisions and refine them may improve effectiveness [92].

Decomposition of existing Verification Techniques. We demonstrated the decomposition of existing verification techniques on the example of CEGAR. More techniques could be decomposed into individual software units to examine the feasibility of decomposition on more examples and to achieve the positive effects of decomposition. Notable mentions are CoVeriTest [40], bounded model checking [13], modular verification [20, 92], and k-induction [51, 81].

5.2 Conclusion

The presented research lays the groundwork for large-scale experimental comparisons in test generation for C programs and demonstrates the potential of cooperation in both formal verification, test generation, and in-between.

(1) We were the first to do large-scale experimental comparisons of formal verifiers and test generators. Succinctly, we created techniques for Test-Comp that support reliable experimental comparisons of test generators for C programs. Last, we closed the gap between formal verifiers and test generators by presenting a technique that turns any formal verifier of SV-COMP into a directed test generator.

(2) We explored the potential of cooperative software verification in different directions. First, we made cooperation with conditions widely usable by encoding the proprietary condition-automaton format as program code. Then, we introduced conditional testing and enabled to turn any formal verifier into a test generator for coverage criteria. On the example of incremental verification, we showed the flexibility of conditions. Last, we decomposed CEGAR into stand-alone components to demonstrate how existing techniques for software verification can be decoupled.

Our work provides a fundamental contribution towards cooperative software verification with automated test generation and automated formal verification, and, to this end, a fundamental contribution to automated software verification in general.

Availability. Whenever feasible, our research is published with open access. We provide reproduction packages and the raw data of our experiments for each article.

We provide multiple links to web pages in this thesis. Because things on the internet rarely last forever, there are two ways to access archived versions of the web pages: (1) If the URL is to a software repository on `github.com` or `gitlab.com`, use the Software Heritage Archive: enter the URL of the repository in the search field at `https://archive.softwareheritage.org/`. For example, for the original software repository `https://gitlab.com/sosy-lab/software/cpachecker/`, you can access its archive through `https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.com/sosy-lab/software/cpachecker.git`. (2) For all other URLs, use the Internet Archive Wayback Machine: prefix the original URL with `https://web.archive.org/web/` to get the latest archived web-page version. For example, for the original URL `https://test-comp.sosy-lab.org/2022/`, you can access its archive through `https://web.archive.org/web/https://test-comp.sosy-lab.org/2022/`.

The source of this thesis is available at `https://gitlab.com/lemberger/phd-thesis`.

Bibliography

[1] Aaron R. Bradley. “SAT-Based model checking without unrolling”. In: *Proc. VMCAI*. LNCS 6538. Springer, 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[2] Ákos Hajdu and Zoltán Micskei. “Efficient Strategies for CEGAR-Based Model Checking”. In: *J. Autom. Reasoning* 64.6 (2020), pp. 1051–1091. DOI: 10.1007/s10817-019-09535-x.

[3] Alan Turing. “Checking a Large Routine”. In: *Report on a Conference on High Speed Automatic Calculating Machines*. Cambridge Univ. Math. Lab., 1949, pp. 67–69.

[4] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. “Software Verification Using k-Induction”. In: *Proc. SAS*. LNCS 6887. Springer, 2011, pp. 351–368. DOI: 10.1007/978-3-642-23702-7_26.

[5] Alex Denisov and Stanislav Pankevich. “Mull It Over: Mutation Testing Based on LLVM”. In: *Proc. ICST*. IEEE, 2018, pp. 25–31. DOI: 10.1109/ICSTW.2018.00024.

[6] Alexander Knüppel, Thomas Thüm, and Ina Schaefer. “GUIDO: Automated Guidance for the Configuration of Deductive Program Verifiers”. In: *Proc. FormaliSE@ICSE*. IEEE, 2021, pp. 124–129. DOI: 10.1109/FormaliSE52586.2021.00018.

[7] Allen T. Acree, Timothy A. Budd, Richard J. Lipton, Richard A. DeMillo, and Frederick G. Sayward. *Mutation Analysis*. Tech. rep. YALEU/DCS/TR155. Yale University, Apr. 1979.

[8] Anders Eklund, Thomas E. Nichols, and Hans Knutsson. “Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates”. In: *Proceedings of the National Academy of Sciences* 113.28 (2016), pp. 7900–7905. DOI: 10.1073/pnas.1602413113.

[9] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. “FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement”. In: *Proc. CAV*. LNCS 5123. Springer, 2008, pp. 209–213. DOI: 10.1007/978-3-540-70545-1_20.

[10] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. "How did you specify your test suite". In: *Proc. ASE*. ACM, 2010, pp. 407–416. DOI: 10.1145/1858996.1859084.

[11] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. "Query-Driven Program Testing". In: *Proc. VMCAI*. LNCS 5403. Springer, 2009, pp. 151–166. DOI: 10.1007/978-3-540-93900-9_15.

[12] Antoine Miné. "The Octagon Abstract Domain". In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1.

[13] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *Proc. TACAS*. LNCS 1579. Springer, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[14] Arthur G. Stephenson, Daniel R. Mulville, Frank H. Bauer, Greg A. Dukeman, Peter Norvig, Lia S. LaPiana, Peter J. Rutledge, David Folta, and Robert Sackheim. *Mars Climate Orbiter Mishap Investigation Board. Phase I Report*. Tech. rep. Nov. 1999.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking". In: *Advances in Computers* 58 (2003), pp. 117–148. DOI: 10.1016/S0065-2458(03)58003-2.

[16] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". In: *Proc. ICSE*. IEEE, 2007, pp. 75–84. DOI: 10.1109/ICSE.2007.37.

[17] Cedric Richter and Heike Wehrheim. "Attend and represent: a novel view on algorithm selection for software verification". In: *Proc. ASE*. 2020, pp. 1016–1028. DOI: 10.1145/3324884.3416633.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proc. OSDI*. USENIX Association, 2008, pp. 209–224.

[19] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification". In: *Proc. NFM*. LNCS 9058. Springer, 2015, pp. 3–11. DOI: 10.1007/978-3-319-17524-9_1.

[20] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yan. "Compositional Shape Analysis by Means of Bi-Abduction". In: *J. ACM* 58.6 (2011), 26:1–26:66. DOI: 10.1145/2049697.2049700.

[21] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. "Craig vs. Newton in software model checking". In: *Proc. ESEC/FSE*. ACM, 2017, pp. 487–497. DOI: 10.1145/3106237.3106307.

[22] Dirk Beyer. “Advances in Automatic Software Testing: Test-Comp 2022”. In: *Proc. FASE*. LNCS 13241. Springer, 2022, pp. 321–335. DOI: 10.1007/978-3-030-99429-7_18.

[23] Dirk Beyer. “Automatic Verification of C and Java Programs: SV-COMP 2019”. In: *Proc. TACAS (3)*. LNCS 11429. Springer, 2019, pp. 133–155. DOI: 10.1007/978-3-030-17502-3_9.

[24] Dirk Beyer. “Competition on Software Verification and Witness Validation: SV-COMP 2023”. In: *Proc. TACAS (2)*. LNCS 13994. Springer, 2023, pp. 495–522. DOI: 10.1007/978-3-031-30820-8_29.

[25] Dirk Beyer. “First International Competition on Software Testing (Test-Comp 2019)”. In: *Int. J. Softw. Tools Technol. Transf.* 23.6 (Dec. 2021), pp. 833–846. DOI: 10.1007/s10009-021-00613-3.

[26] Dirk Beyer. “Progress on Software Verification: SV-COMP 2022”. In: *Proc. TACAS (2)*. LNCS 13244. Springer, 2022, pp. 375–402. DOI: 10.1007/978-3-030-99527-0_20.

[27] Dirk Beyer. “Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)”. In: *Proc. TACAS*. LNCS 9636. Springer, 2016, pp. 887–904. DOI: 10.1007/978-3-662-49674-9_55.

[28] Dirk Beyer. *Results of the 12th Intl. Competition on Software Verification (SV-COMP 2023)*. Zenodo. 2023. DOI: 10.5281/zenodo.7627787.

[29] Dirk Beyer. *Results of the 5th Intl. Competition on Software Testing (Test-Comp 2023)*. Zenodo. 2023. DOI: 10.5281/zenodo.7701122.

[30] Dirk Beyer. “Software Verification and Verifiable Witnesses (Report on SV-COMP 2015)”. In: *Proc. TACAS*. LNCS 9035. Springer, 2015, pp. 401–416. DOI: 10.1007/978-3-662-46681-0_31.

[31] Dirk Beyer. “Software Verification with Validation of Results (Report on SV-COMP 2017)”. In: *Proc. TACAS*. LNCS 10206. Springer, 2017, pp. 331–349. DOI: 10.1007/978-3-662-54580-5_20.

[32] Dirk Beyer. “Status Report on Software Testing: Test-Comp 2021”. In: *Proc. FASE*. LNCS 12649. Springer, 2021, pp. 341–357. DOI: 10.1007/978-3-030-71500-7_17.

[33] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. “Generating Tests from Counterexamples”. In: *Proc. ICSE*. IEEE, 2004, pp. 326–335. DOI: 10.1109/ICSE.2004.1317455.

[34] Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. “Information Reuse for Multi-goal Reachability Analyses”. In: *Proc. ESOP*. LNCS 7792. Springer, 2013, pp. 472–491. DOI: 10.1007/978-3-642-37036-6_26.

[35] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. “Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR”. In: *Proc. ICSE*. ACM, 2022.

[36] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. *Reproduction Package (VM Version) for ICSE 2022 Article 'Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR'*. Zenodo. 2022. DOI: 10.5281/zenodo.5918111.

[37] Dirk Beyer and Karlheinz Friedberger. "Domain-Independent Interprocedural Program Analysis using Block-Abstraction Memoization". In: *Proc. ESEC/FSE*. ACM, 2020, pp. 50–62. DOI: 10.1145/3368089.3409718.

[38] Dirk Beyer and Karlheinz Friedberger. "Domain-Independent Multi-threaded Software Model Checking". In: *Proc. ASE*. ACM, 2018, pp. 634–644. DOI: 10.1145/3238147.3238195.

[39] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Proc. CAV*. LNCS 6806. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[40] Dirk Beyer and Marie-Christine Jakobs. "Cooperative Verifier-Based Testing with CoVeriTest". In: *Int. J. Softw. Tools Technol. Transfer* 23.3 (2021), pp. 313–333. DOI: 10.1007/s10009-020-00587-8.

[41] Dirk Beyer and Marie-Christine Jakobs. "FRed: Conditional Model Checking via Reducers and Folders". In: *Proc. SEFM*. LNCS 12310. Springer, 2020, pp. 113–132. DOI: 10.1007/978-3-030-58768-0_7.

[42] Dirk Beyer, Marie-Christine Jakobs, and Thomas Lemberger. "Difference Verification with Conditions". In: *Proc. SEFM*. LNCS 12310. Springer, 2020, pp. 133–154. DOI: 10.1007/978-3-030-58768-0_8.

[43] Dirk Beyer, Marie-Christine Jakobs, and Thomas Lemberger. *Reproduction Package for Article 'Difference Verification with Conditions'*. Zenodo. 2020. DOI: 10.5281/zenodo.3954933.

[44] Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. "Reducer-Based Construction of Conditional Verifiers". In: *Proc. ICSE*. ACM, 2018, pp. 1182–1193. DOI: 10.1145/3180155.3180259.

[45] Dirk Beyer and Martin Spiessl. "MetaVal: Witness Validation via Verification". In: *Proc. CAV*. LNCS 12225. Springer, 2020, pp. 165–177. DOI: 10.1007/978-3-030-53291-8_10.

[46] Dirk Beyer and Matthias Dangl. "Strategy Selection for Software Verification Based on Boolean Features: A Simple but Effective Approach". In: *Proc. ISoLA*. LNCS 11245. Springer, 2018, pp. 144–159. DOI: 10.1007/978-3-030-03421-4_11.

[47] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. "Correctness Witnesses: Exchanging Verification Results Between Verifiers". In: *Proc. FSE*. ACM, 2016, pp. 326–337. DOI: 10.1145/2950290.2950351.

[48] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. "Witness Validation and Stepwise Testification across Software Verifiers". In: *Proc. FSE*. ACM, 2015, pp. 721–733. DOI: 10.1145/2786805.2786867.

[49] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. "Verification Witnesses". In: *ACM Trans. Softw. Eng. Methodol.* (2022). DOI: 10.1145/3477579.

[50] Dirk Beyer, Matthias Dangl, and Philipp Wendler. "A Unifying View on SMT-Based Software Verification". In: *J. Autom. Reasoning* 60.3 (2018), pp. 299–335. ISSN: 1573-0670. DOI: 10.1007/s10817-017-9432-6.

[51] Dirk Beyer, Matthias Dangl, and Philipp Wendler. "Boosting k-Induction with Continuously-Refined Invariants". In: *Proc. CAV*. LNCS 9206. Springer, 2015, pp. 622–640. DOI: 10.1007/978-3-319-21690-4_42.

[52] Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig. "Tests from Witnesses: Execution-Based Validation of Verification Results". In: *Proc. TAP*. LNCS 10889. Springer, 2018, pp. 3–23. DOI: 10.1007/978-3-319-92994-1_1.

[53] Dirk Beyer and Philipp Wendler. "CPU Energy Meter: A Tool for Energy-Aware Algorithms Engineering". In: *Proc. TACAS (2)*. LNCS 12079. Springer, 2020, pp. 126–133. DOI: 10.1007/978-3-030-45237-7_8.

[54] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable Benchmarking: Requirements and Solutions". In: *Int. J. Softw. Tools Technol. Transfer* 21.1 (2019), pp. 1–29. DOI: 10.1007/s10009-017-0469-y.

[55] Dirk Beyer and Sudeep Kanav. "CoVeriTeam: On-Demand Composition of Cooperative Verification Systems". In: *Proc. TACAS*. LNCS 13243. Springer, 2022, pp. 561–579. DOI: 10.1007/978-3-030-99524-9_31.

[56] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. "Conditional Model Checking: A Technique to Pass Information between Verifiers". In: *Proc. FSE*. ACM, 2012. DOI: 10.1145/2393596.2393664.

[57] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The Software Model Checker Blast". In: *Int. J. Softw. Tools Technol. Transfer* 9.5-6 (2007), pp. 505–525. DOI: 10.1007/s10009-007-0044-z.

[58] Dirk Beyer and Thomas Lemberger. "Conditional Testing: Off-the-Shelf Combination of Test-Case Generators". In: *Proc. ATVA*. LNCS 11781. Springer, 2019, pp. 189–208. DOI: 10.1007/978-3-030-31784-3_11.

[59] Dirk Beyer and Thomas Lemberger. *Replication Package for Article 'Software Verification: Testing vs. Model Checking", Proc. HVC '17*. 2018. DOI: 10.5281/zenodo.1158646.

[60] Dirk Beyer and Thomas Lemberger. *Reproduction Package for Article 'Five Years Later: Testing vs. Model Checking'*. Zenodo. 2022.

[61] Dirk Beyer and Thomas Lemberger. *Reproduction Package for Article 'TestCov: Robust Test-Suite Execution and Coverage Measurement' in Proc. ASE '19*. Zenodo. 2019. DOI: 10.5281/zenodo.3418726.

[62] Dirk Beyer and Thomas Lemberger. *Reproduction Package for ATVA 2019 Article 'Conditional Testing: Off-the-Shelf Combination of Test-Case Generators'*. 2019. DOI: 10.5281/zenodo.3352401.

[63] Dirk Beyer and Thomas Lemberger. "Six Years Later: Testing vs. Model Checking". In: *Int. J. Softw. Tools Technol. Transfer* (2023). Under review.

[64] Dirk Beyer and Thomas Lemberger. "Software Verification: Testing vs. Model Checking". In: *Proc. HVC*. LNCS 10629. Springer, 2017, pp. 99–114. DOI: 10.1007/978-3-319-70389-3_7.

[65] Dirk Beyer and Thomas Lemberger. "Symbolic Execution with CEGAR". In: *Proc. ISoLA*. LNCS 9952. Springer, 2016, pp. 195–211. DOI: 10.1007/978-3-319-47166-2_14.

[66] Dirk Beyer and Thomas Lemberger. "TestCov: Robust Test-Suite Execution and Coverage Measurement". In: *Proc. ASE*. IEEE, 2019, pp. 1074–1077. DOI: 10.1109/ASE.2019.00105.

[67] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. "LEGION: Best-First Concolic Testing". In: *Proc. ASE*. IEEE, 2020, pp. 54–65. DOI: 10.1145/3324884.3416629.

[68] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *J. ACM* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[69] Edmund Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

[70] Edsger W. Dijkstra. "The Humble Programmer". In: *Commun. ACM* 15.10 (1972), pp. 859–866. DOI: 10.1145/355604.361591.

[71] Farah Hariri and August Shi. "SRCIROR: a toolset for mutation testing of C source code and LLVM intermediate representation". In: *Proc. ASE*. ACM, 2018, pp. 860–863. DOI: 10.1145/3238147.3240482.

[72] Gordon Fraser and Andrea Arcuri. "EvoSuite: automatic test suite generation for object-oriented software". In: *Proc. ESEC/FSE*. ACM, 2011, pp. 416–419. DOI: 10.1145/2025113.2025179.

[73] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119. DOI: 10.1561/2500000010.

[74] Arie Gurfinkel and Sagar Chaki. "Boxes: A Symbolic Abstract Domain of Boxes". In: *Proc. SAS*. 2010, pp. 287–303. DOI: 10.1007/978-3-642-15769-1_18.

[75] Reiner Hähnle and Marieke Huisman. "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools". In: *Computing and Software Science—State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard J. Woeginger. LNCS 10000. Springer, 2019, pp. 345–373. DOI: 10.1007/978-3-319-91908-9_18.

[76] Hoang M. Le. "LLVM-Based Hybrid Fuzzing with LIBKLUZZER (Competition Contribution)". In: *Proc. FASE*. LNCS 12076. Springer, 2020, pp. 535–539. DOI: 10.1007/978-3-030-45234-6_29.

[77] Jacob Burnim and Koushik Sen. "Heuristics for Scalable Dynamic Test Generation". In: *Proc. ASE*. IEEE, 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69.

[78] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. *Ariane 501 Inquiry Board Report*. Tech. rep. July 1996.

[79] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[80] James H. Andrews, Lionel C. Briand, and Yvan Labiche. "Is mutation an appropriate tool for testing experiments?" In: *Proc. ICSE*. ACM, 2005, pp. 402–411. DOI: 10.1145/1062455.1062530.

[81] Jan Haltermann and Heike Wehrheim. "CoVEGI: Cooperative Verification via Externally Generated Invariants". In: *Proc. FASE*. LNCS 12649. 2021, pp. 108–129. DOI: 10.1007/978-3-030-71500-7_6.

[82] Susmit Jha and Sanjit A. Seshia. "A theory of formal synthesis via inductive learning". In: *Acta Informatica* 54.7 (2017), pp. 693–726. DOI: 10.1007/s00236-017-0294-5.

[83] Jiří Barnat, Jakub Havlícek, and Petr Ročkai. "Distributed LTL Model Checking with Hash Compaction". In: *ENTCS* 296 (2013), pp. 79–93. DOI: 10.1016/j.entcs.2013.07.006.

[84] Jiří Slabý, Jan Strejček, and Marek Trtík. "Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution". In: *Proc. FMICS*. LNCS 7437. Springer, 2012, pp. 207–221. DOI: 10.1007/978-3-642-32469-7_14.

[85] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. "TRACER: A Symbolic Execution Tool for Verification". In: *Proc. CAV*. LNCS 7358. Springer, 2012, pp. 758–766. DOI: 10.1007/978-3-642-31424-7_61.

[86] Kaled Alshmrany, Mohannad Aldughaim, Lucas C. Cordeiro, and Ahmed Bhayat. "FUSEBMC V.4: Smart Seed Generation for Hybrid Fuzzing (Competition Contribution)". In: *Proc. FASE*. LNCS 13241. Springer, 2022, pp. 336–340. DOI: 10.1007/978-3-030-99429-7_19.

[87] Kaled M. Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C. Cordeiro. "FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs". In: *Proc. TAP*. LNCS 12740. Springer, 2021, pp. 85–105. DOI: 10.1007/978-3-030-79379-1_6.

[88] Kalpesh Kapoor and Jonathan P. Bowen. "A formal analysis of MCDC and RCDC test criteria". In: *Softw. Test. Verification Reliab.* 15.1 (2005), pp. 21–40. DOI: 10.1002/stvr.306.

[89] Yunho Kim, Zhihong Xu, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. "Hybrid Directed Test Suite Augmentation: An Interleaving Framework". In: *Proc. ICST*. IEEE, 2014, pp. 263–272. DOI: 10.1109/ICST.2014.39.

[90] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. "Directed Symbolic Execution". In: *Proc. SAS*. LNCS 6887. Springer, 2011, pp. 95–111. DOI: 10.1007/978-3-642-23702-7_11.

[91] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proc. FSE*. ACM, 2005, pp. 263–272. DOI: 10.1145/1081706.1081750.

[92] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even-Mendoza, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. "HiFrog: SMT-Based Function Summarization for Software Verification". In: *Proc. TACAS*. LNCS 10206. 2017, pp. 207–213. DOI: 10.1007/978-3-662-54580-5_12.

[93] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: A survey". In: *Cybersecurity* 1.1 (June 2018), p. 6. ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y.

[94] Zohar Manna and Richard J. Waldinger. "A Deductive Approach to Program Synthesis". In: *ACM Trans. Program. Lang. Syst.* 2.1 (1980), pp. 90–121. DOI: 10.1145/357084.357090.

[95] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *Proc. SIGSAC*. New York, NY, USA: ACM, 2016, pp. 1032–1043. DOI: 10.1145/2976749.2978428.

[96] Marek Chalupa, Anna Řechtáčková, Vincent Mihalkovič, Lukáš Zaoral, and Jan Strejček. "SYMBIOTIC 9: String Analysis and Backward Symbolic Execution with Loop Folding (Competition Contribution)". In: *Proc. TACAS (2)*. LNCS 13244. Springer, 2022, pp. 462–467. DOI: 10.1007/978-3-030-99527-0_32.

[97] Maria Christakis, Peter Müller, and Valentin Wüstholz. "Collaborative Verification and Testing with Explicit Assumptions". In: *Proc. FM*. LNCS 7436. Springer, 2012, pp. 132–146. DOI: 10.1007/978-3-642-32759-9_13.

[98] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. "Testability Transformation". In: *IEEE Trans. Software Eng.* 30.1 (2004), pp. 3–16. DOI: 10.1109/TSE.2004.1265732.

[99] Mark Weiser. "Program Slicing". In: *IEEE Trans. Softw. Eng.* 10.4 (1984), pp. 352–357. DOI: 10.1109/tse.1984.5010248.

[100] Matthias Dangl, Stefan Löwe, and Philipp Wendler. "CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic (Competition Contribution)". In: *Proc. TACAS*. LNCS 9035. Springer, 2015, pp. 423–425. DOI: 10.1007/978-3-662-46681-0_34.

[101] Kenneth L. McMillan. "Interpolation and Model Checking". In: *Handbook of Model Checking*. Springer, 2018, pp. 421–446. DOI: 10.1007/978-3-319-10575-8_14.

[102] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. "Just Test What You Cannot Verify!" In: *Proc. FASE*. LNCS 9033. Springer, 2015, pp. 100–114. DOI: 10.1007/978-3-662-46675-9_7.

[103] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. "Mutation Testing Advances: An Analysis and Survey". In: *Adv. Comput.* 112 (2019), pp. 275–378. DOI: 10.1016/bs.adcom.2018.03.015.

[104] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. "What It Would Take to Use Mutation Testing in Industry - A Study at Facebook". In: *Proc. ICSE (SEIP)*. IEEE, 2021, pp. 268–277. DOI: 10.1109/ICSE-SEIP52600.2021.00036.

[105] Nancy G. Leveson and Clark S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[106] Oscar S. Dustmann, Klaus Wehrle, and Cristian Cadar. "PARTI: a multi-interval theory solver for symbolic execution". In: *Proc. ASE*. ACM, 2018, pp. 430–440. DOI: 10.1145/3238147.3238179.

[107] Pankai Jalote, Vipindeep Vangala, Taranbir Singh, and Prateek Jain. "Program Partitioning: A Framework for Combining Static and Dynamic Analysis". In: *Proc. WODA*. Shanghai, China: ACM, 2006, pp. 11–16. DOI: 10.1145/1138912.1138916.

[108] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "Dart: Directed Automated Random Testing". In: *Proc. PLDI*. ACM, 2005, pp. 213–223. DOI: 10.1145/1065010.1065036.

[109] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. "Compositional May-must Program Analysis: Unleashing the Power of Alternation". In: *Proc. POPL*. ACM, 2010, pp. 43–56. DOI: 10.1145/1706299.1706307.

[110] Patrick Cousot and Radhia Cousot. "Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints". In: *Proc. POPL*. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[111] Nir Piterman and Amir Pnueli. "Temporal Logic and Fair Discrete Systems". In: *Handbook of Model Checking*. Springer, 2018, pp. 27–73. DOI: 10.1007/978-3-319-10575-8_2.

[112] Przemysław Daca, Ashutosh Gupta, and Thomas A. Henzinger. “Abstraction-Driven Concolic Testing”. In: *Proc. VMCAI.* LNCS 9583. Springer, 2016, pp. 328–347. DOI: 10.1007/978-3-662-49122-5_16.

[113] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. “Synthesis Through Unification”. In: *Proc. CAV.* LNCS 9207. Springer, 2015, pp. 163–179. DOI: 10.1007/978-3-319-21668-3_10.

[114] Ravindra Metta, Raveendra Kumar Medicherla, and Hrishikesh Karmarkar. “VeriFuzz: Good Seeds for Fuzzing (Competition Contribution)”. In: *Proc. FASE.* LNCS 13241. Springer, 2022, pp. 341–346. DOI: 10.1007/978-3-030-99429-7_20.

[115] Rupak Majumdar and Koushik Sen. “Hybrid Concolic Testing”. In: *Proc. ICSE.* IEEE, 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41.

[116] Sebastian Ruland, Malte Lochau, and Marie-Christine Jakobs. “HybridTiger: Hybrid Model Checking and Domination-Based Partitioning for Efficient Multi-Goal Test-Suite Generation (Competition Contribution)”. In: *Proc. FASE.* LNCS 12076. Springer, 2020, pp. 520–524. DOI: 10.1007/978-3-030-45234-6_26.

[117] Shikhar Singh and Sarfraz Khurshid. “Distributed Symbolic Execution using Test-Depth Partitioning”. In: *CoRR* abs/2106.02179 (2021). arXiv: 2106.02179.

[118] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. “From program verification to program synthesis”. In: *Proc. POPL.* Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 313–326. DOI: 10.1145/1706299.1706337.

[119] Stefan Löwe, Mikhail U. Mandrykin, and Philipp Wendler. “CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses (Competition Contribution)”. In: *Proc. TACAS.* LNCS 8413. Springer, 2014, pp. 392–394. DOI: 10.1007/978-3-642-54862-8_27.

[120] Susan Horwitz, Thomas W. Reps, and David W. Binkley. “Interprocedural Slicing Using Dependence Graphs”. In: *ACM Trans. Program. Lang. Syst.* 12.1 (1990), pp. 26–60. DOI: 10.1145/77606.77608.

[121] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. “Mart: a mutant generation tool for LLVM”. In: *Proc. ESEC/FSE.* ACM, 2019, pp. 1080–1084. DOI: 10.1145/3338906.3341180.

[122] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. “Abstractions from proofs”. In: *Proc. POPL.* ACM, 2004, pp. 232–244. DOI: 10.1145/964001.964021.

[123] Thomas Ball, Rupak Majumdar, T. D. Millstein, and S. K. Rajamani. “Automatic Predicate Abstraction of C Programs”. In: *Proc. PLDI.* ACM, 2001, pp. 203–213. DOI: 10.1145/378795.378846.

[124] Thomas Lemberger. “Plain Random Test Generation with PRTest (Competition Contribution)”. In: *Int. J. Softw. Tools Technol. Transf.* 23.6 (Dec. 2021), pp. 871–873. DOI: 10.1007/s10009-020-00568-x.

[125] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. “GraphML Progress Report”. In: *Graph Drawing*. LNCS 2265. Springer, 2001, pp. 501–512. DOI: 10.1007/3-540-45848-4_59.

[126] Will Leeson and Matthew B. Dwyer. “Algorithm Selection for Software Verification using Graph Attention Networks”. In: *CoRR* abs/2201.11711 (2022). arXiv: 2201.11711.

[127] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. “Model Checking Programs”. In: *Autom. Software Eng.* 10.2 (2003), pp. 203–232. DOI: 10.1023/A:1022920129859.

[128] William E. Howden. “Weak Mutation Testing and Completeness of Test Sets”. In: *IEEE Trans. Software Eng.* 8.4 (1982), pp. 371–379. DOI: 10.1109/TSE.1982.235571.

[129] William H. Pickering. *Mariner-Venus 1962. Final Project Report.* Tech. rep. NASA SP-59. July 1962.

[130] Yavuz Köroglu and Alper Sen. “Design of a Modified Concolic Testing Algorithm with Smaller Constraints”. In: *Proc. CSTVA@ISSTA*. CEUR 1639. CEUR-WS.org, 2016, pp. 3–14.

[131] Yibiao Yang, Yanyan Jiang, Zhiqiang Zuo, Yang Wang, Hao Sun, Hongmin Lu, Yuming Zhou, and Baowen Xu. “Automatic Self-Validation for Code Coverage Profilers”. In: *Proc. ASE*. IEEE, 2019, pp. 79–90. DOI: 10.1109/ASE.2019.00018.

[132] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. “The Matter of Heartbleed”. In: *Proc. IMC*. ACM, 2014, pp. 475–488. DOI: 10.1145/2663716.2663755.

A Credits

Software Verification: Testing vs. Model Checking

This article (Section B.1) is authored by Dirk Beyer and Thomas Lemberger and published by Springer in the proceedings of HVC 2017 [64]. The article received the best paper award at HVC 2017. A reproduction package [59] is available and the tool TBF is available open source. Thomas Lemberger is a co-author to this article and contributed about 70 % of the article's content.

Plain Random Test Generation with PRTest (Competition Contribution)

This article (Section B.2) is authored by Thomas Lemberger and published by Springer in the Journal on Software Tools for Technology Transfer (STTT), 2019 [124]. Reproduction is possible through Test-Comp 2019 [25] and the tool PRTest is available open source. Thomas Lemberger is the sole author of this article, so he contributed 100 % of the article's content.

TestCov: Robust Test-Suite Execution and Coverage Measurement

This article (Section B.3) is authored by Dirk Beyer and Thomas Lemberger and published by IEEE in the proceedings of ASE 2019 [66]. A reproduction package [61] is available, and the tool TestCov is available open source. Thomas Lemberger is a co-author to this article and contributed about 90 % of the article's content.

Tests from Witnesses: Execution-Based Validation of Verification Results

This article (Section B.4) is authored by Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig. The article is published by Springer in the proceedings of TAP 2018 [52]. All experimental data is available[1] and the tools CPA-w2t and FShell-w2t are both available open source. Thomas Lemberger is a co-author to this article and contributed about 40 % of the article's content.

[1] https://www.sosy-lab.org/research/executionbasedwitnessvalidation/

Six Years Later: Testing vs. Model Checking

This article (Section B.5) is authored by Dirk Beyer and Thomas Lemberger. It is submitted to the International Journal on Software Tools for Technology Transfer (STTT) and currently under review. A reproduction package [60] is available. Thomas Lemberger is a co-author to this article and contributed about 90 % of the article's content.

Reducer-Based Construction of Conditional Verifiers

This article (Section B.6) is authored by Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. The article is published by ACM in the proceedings of ICSE 2018 [44]. The concept of reducer is implemented in CPAchecker, which is available open source. Thomas Lemberger is a co-author to this article and contributed about 30 % of the article's content.

Conditional Testing: Off-the-Shelf Combination of Test-Case Generators

This article (Section B.7) is authored by Dirk Beyer and Thomas Lemberger. The article is published by Springer in the proceedings of ATVA 2019 [58]. A reproduction package [62] is available and the software CondTest is available open source. Thomas Lemberger is a co-author to this article and contributed about 80 % of the article's content.

Difference Verification with Conditions

This article (Section B.8) is authored by Dirk Beyer, Marie-Christine Jakobs, and Thomas Lemberger. It is published by Springer in the proceedings of SEFM 2020 [42]. A reproduction package [43] is available and the approach is implemented open source in CPAchecker. Thomas Lemberger is a co-author of this article and contributed about 40 % of the article's content.

Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR

This article (Section B.9) is authored by Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. The article will be published by Springer in the proceedings of ICSE 2022 [35]. A reproduction package [36] is available and our implementation is available open source. Thomas Lemberger is a co-author of this article and contributed about 40 % of the article's content.

B Original Manuscripts

This appendix includes all publications discussed in Chapters 3 and 4. Publications are ordered according to their appearance in this work. In all of the listed publications, author names are in alphabetical order.

Software Verification: Testing vs. Model Checking

A Comparative Evaluation of the State of the Art

Dirk Beyer and Thomas Lemberger

LMU Munich, Germany

Abstract. In practice, software testing has been the established method for finding bugs in programs for a long time. But in the last 15 years, software model checking has received a lot of attention, and many successful tools for software model checking exist today. We believe it is time for a careful comparative evaluation of automatic software testing against automatic software model checking. We chose six existing tools for automatic test-case generation, namely AFL-fuzz, CPATiger, Crest-ppc, FShell, Klee, and PRtest, and four tools for software model checking, namely Cbmc, CPA-Seq, Esbmc-incr, and Esbmc-kInd, for the task of finding specification violations in a large benchmark suite consisting of 5 693 C programs. In order to perform such an evaluation, we have implemented a framework for test-based falsification (TBF) that executes and validates test cases produced by test-case generation tools in order to find errors in programs. The conclusion of our experiments is that software model checkers can (i) find a substantially larger number of bugs (ii) in less time, and (iii) require less adjustment to the input programs.

1 Introduction

Software testing has been the standard technique for identifying software bugs for decades. The exhaustive and sound alternative, software model checking, is believed to be immature for practice. Some often-named disadvantages are the need for experts in formal verification, extreme resource consumption, and maturity issues when it comes to handling large software systems.

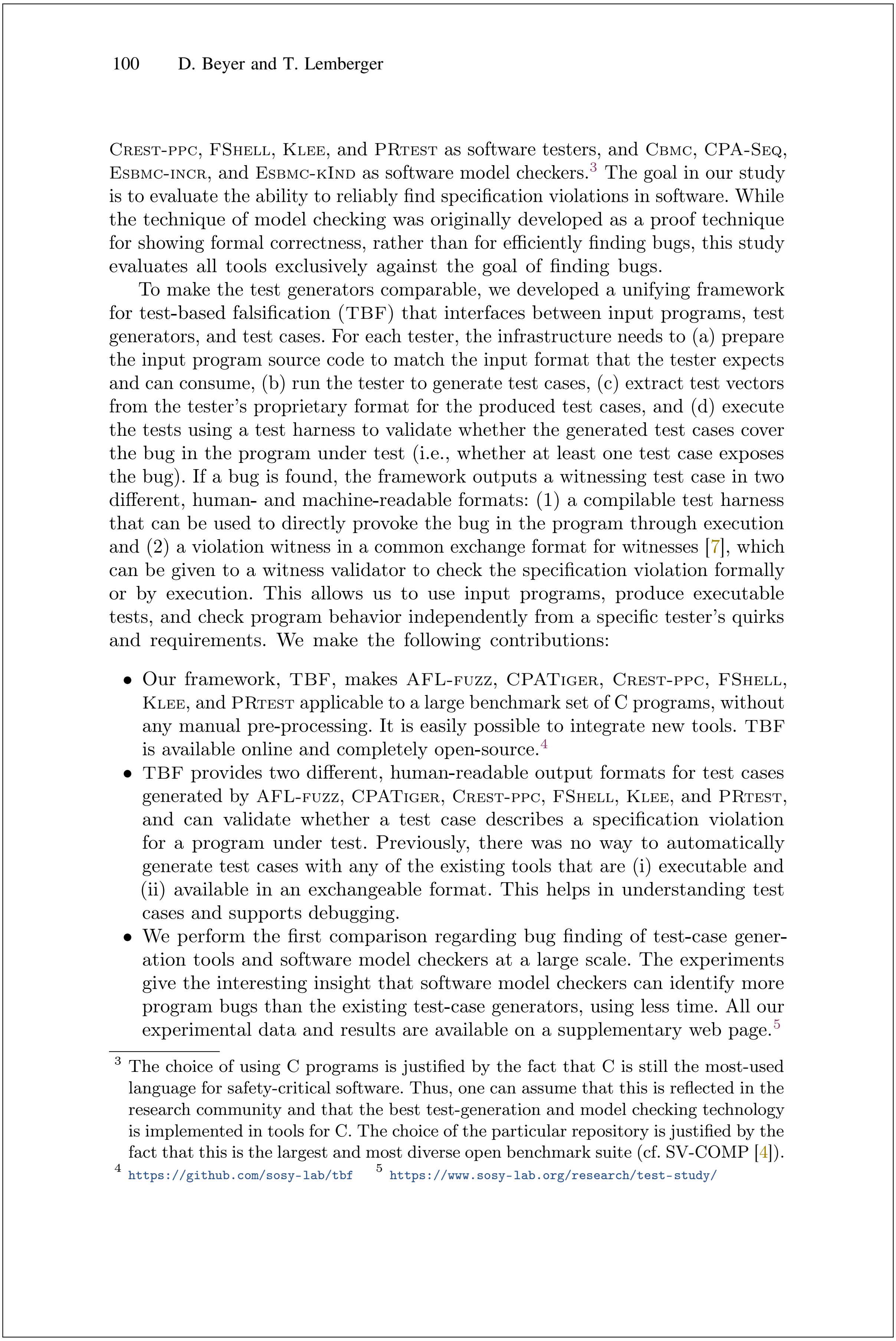
But are these concerns still true today? We claim that the answer is No, and show with experiments on a large benchmark of C programs that software model checkers even find more bugs than testers. We found it is time for a comparative evaluation of testing tools against model-checking tools, motivated by the success of software model checkers as demonstrated in the annual International Competition on Software Verification (SV-COMP) [4], and by the move of development groups of large software systems towards formal verification, such as Facebook[1], Microsoft [2, 44], and Linux [38].

Our contribution is a thorough experimental comparison of software testers against software model checkers. We performed our experimental study on 5 693 programs from a widely-used and state-of-the-art benchmarking set.[2] To represent the state of the art in terms of tools, we use AFL-fuzz, CPATiger,

[1] http://fbinfer.com/ [2] https://github.com/sosy-lab/sv-benchmarks

© Springer International Publishing AG 2017

O. Strichman and R. Tzoref-Brill (Eds.): HVC 2017, LNCS 10629, pp. 99–114, 2017.
https://doi.org/10.1007/978-3-319-70389-3_7



Crest-ppc, FShell, Klee, and PRtest as software testers, and Cbmc, CPA-Seq, Esbmc-incr, and Esbmc-kInd as software model checkers.[3] The goal in our study is to evaluate the ability to reliably find specification violations in software. While the technique of model checking was originally developed as a proof technique for showing formal correctness, rather than for efficiently finding bugs, this study evaluates all tools exclusively against the goal of finding bugs.

To make the test generators comparable, we developed a unifying framework for test-based falsification (TBF) that interfaces between input programs, test generators, and test cases. For each tester, the infrastructure needs to (a) prepare the input program source code to match the input format that the tester expects and can consume, (b) run the tester to generate test cases, (c) extract test vectors from the tester's proprietary format for the produced test cases, and (d) execute the tests using a test harness to validate whether the generated test cases cover the bug in the program under test (i.e., whether at least one test case exposes the bug). If a bug is found, the framework outputs a witnessing test case in two different, human- and machine-readable formats: (1) a compilable test harness that can be used to directly provoke the bug in the program through execution and (2) a violation witness in a common exchange format for witnesses [7], which can be given to a witness validator to check the specification violation formally or by execution. This allows us to use input programs, produce executable tests, and check program behavior independently from a specific tester's quirks and requirements. We make the following contributions:

- Our framework, TBF, makes AFL-fuzz, CPATiger, Crest-ppc, FShell, Klee, and PRtest applicable to a large benchmark set of C programs, without any manual pre-processing. It is easily possible to integrate new tools. TBF is available online and completely open-source.[4]
- TBF provides two different, human-readable output formats for test cases generated by AFL-fuzz, CPATiger, Crest-ppc, FShell, Klee, and PRtest, and can validate whether a test case describes a specification violation for a program under test. Previously, there was no way to automatically generate test cases with any of the existing tools that are (i) executable and (ii) available in an exchangeable format. This helps in understanding test cases and supports debugging.
- We perform the first comparison regarding bug finding of test-case generation tools and software model checkers at a large scale. The experiments give the interesting insight that software model checkers can identify more program bugs than the existing test-case generators, using less time. All our experimental data and results are available on a supplementary web page.[5]

[3] The choice of using C programs is justified by the fact that C is still the most-used language for safety-critical software. Thus, one can assume that this is reflected in the research community and that the best test-generation and model checking technology is implemented in tools for C. The choice of the particular repository is justified by the fact that this is the largest and most diverse open benchmark suite (cf. SV-COMP [4]).

[4] https://github.com/sosy-lab/tbf

[5] https://www.sosy-lab.org/research/test-study/

Related Work. A large-scale comparative evaluation of the bug-finding capabilities of software testers and software model checkers is missing in the literature and this work is a first contribution towards filling this gap. In the area of software model checking, SV-COMP serves as a yearly comparative evaluation of a large set of model checkers for C programs and the competition report provides an overview over tools and techniques [4]. A general survey over techniques for software model checking is available [37]. In the area of software testing, there is work comparing test-case generators [28]. Surveys provide an overview of different test techniques [1] and a detailed web site is available that provides an overview over tools and techniques [6].

2 Background: Technology and Tools

In this paper, we consider only fully automatic techniques for testing and model checking of whole programs. This means that (i) a verification task consists of a program (with function `main` as entry) and a specification (reduced to reachability of function `__VERIFIER_error` by instrumentation), (ii) the comparison excludes all approaches for partial verification, such as unit testing and procedure summarization, and (iii) the comparison excludes all approaches that require interaction as often needed for deductive verification.

2.1 Software Testing

Given a software system and a specification of that system, testing executes the system with different input values and observes whether the intended behavior is exhibited (i.e., the specification holds). A *test vector* $\langle \eta_1, \cdots, \eta_n \rangle$ is a sequence of n input values η_1 to η_n. A *test case* is described by a test vector, where the i-th input of the test case is given by the i-th value η_i of the test vector. A *test suite* is a set of test cases. A *test harness* is a software that supports the automatic execution of a test case for the program under test, i.e., it feeds the values from the test vector one by one as input to the program. *Test-case generation* produces a set of test vectors that fulfills a specific coverage criterion. Program-branch coverage is an example of a well-established coverage criterion.

There are three major approaches to software test-case generation: symbolic or concolic execution [18, 19, 29, 39, 45, 46], random fuzz testing [30, 36], and model checking [5, 10, 35]. In this work, we use one tester based on symbolic execution (Klee), one based on concolic execution (Crest-ppc), one based on random generation (PRtest), one based on random fuzzing (AFL-fuzz), and two based on model-checking (CPATiger and FShell), which we describe in the following in alphabetic order. Table 1 gives an overview over testers and model checkers.

AFL-fuzz [17] is a coverage-based greybox fuzzer. Given a set of start inputs, it performs different mutations (e.g., bit flips, simple arithmetics) on the existing inputs, executes these newly created inputs, and checks which parts of the program are explored. Depending on these, it decides which inputs to keep, and which to use for further mutations. *Output:* AFL-fuzz outputs each generated

[6] Provided by Z. Micskei: http://mit.bme.hu/~micskeiz/pages/code_based_test_generation.html

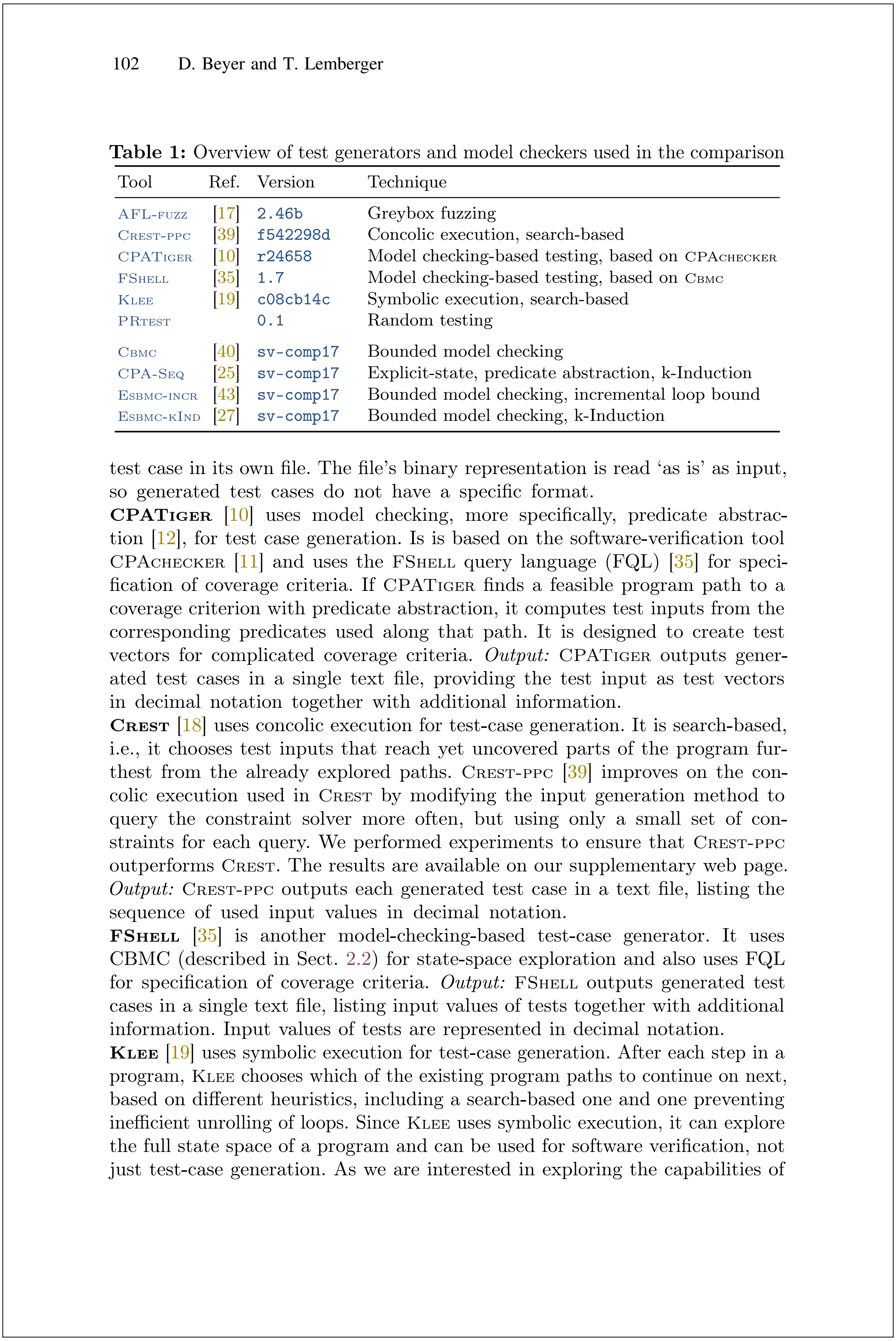


Table 1: Overview of test generators and model checkers used in the comparison

Tool	Ref.	Version	Technique
AFL-fuzz	[17]	`2.46b`	Greybox fuzzing
Crest-ppc	[39]	`f542298d`	Concolic execution, search-based
CPATiger	[10]	`r24658`	Model checking-based testing, based on CPAchecker
FShell	[35]	`1.7`	Model checking-based testing, based on Cbmc
Klee	[19]	`c08cb14c`	Symbolic execution, search-based
PRtest		`0.1`	Random testing
Cbmc	[40]	`sv-comp17`	Bounded model checking
CPA-Seq	[25]	`sv-comp17`	Explicit-state, predicate abstraction, k-Induction
Esbmc-incr	[43]	`sv-comp17`	Bounded model checking, incremental loop bound
Esbmc-kInd	[27]	`sv-comp17`	Bounded model checking, k-Induction

test case in its own file. The file's binary representation is read 'as is' as input, so generated test cases do not have a specific format.

CPATiger [10] uses model checking, more specifically, predicate abstraction [12], for test case generation. Is is based on the software-verification tool CPAchecker [11] and uses the FShell query language (FQL) [35] for specification of coverage criteria. If CPATiger finds a feasible program path to a coverage criterion with predicate abstraction, it computes test inputs from the corresponding predicates used along that path. It is designed to create test vectors for complicated coverage criteria. *Output:* CPATiger outputs generated test cases in a single text file, providing the test input as test vectors in decimal notation together with additional information.

Crest [18] uses concolic execution for test-case generation. It is search-based, i.e., it chooses test inputs that reach yet uncovered parts of the program furthest from the already explored paths. Crest-ppc [39] improves on the concolic execution used in Crest by modifying the input generation method to query the constraint solver more often, but using only a small set of constraints for each query. We performed experiments to ensure that Crest-ppc outperforms Crest. The results are available on our supplementary web page. *Output:* Crest-ppc outputs each generated test case in a text file, listing the sequence of used input values in decimal notation.

FShell [35] is another model-checking-based test-case generator. It uses CBMC (described in Sect. 2.2) for state-space exploration and also uses FQL for specification of coverage criteria. *Output:* FShell outputs generated test cases in a single text file, listing input values of tests together with additional information. Input values of tests are represented in decimal notation.

Klee [19] uses symbolic execution for test-case generation. After each step in a program, Klee chooses which of the existing program paths to continue on next, based on different heuristics, including a search-based one and one preventing inefficient unrolling of loops. Since Klee uses symbolic execution, it can explore the full state space of a program and can be used for software verification, not just test-case generation. As we are interested in exploring the capabilities of

testing, we only consider the test cases produced by Klee. *Output:* Klee outputs each generated test case in a binary format that can be read with Klee. The input values of tests are represented by their bit width and bit representation.
PRtest is a simple tool for plain random testing. The tool is delivered together with TBF and serves as base line in our experiments. *Output:* PRtest outputs each generated test case in a text file, listing the sequence of used input values in hexadecimal notation.

2.2 Software Model Checking

Software model checking tries to prove a program correct or find a property violation in a program, by exploring the full state space and checking whether any of the feasible program states violate the specification. A lot of different techniques exist to do this. Since the number of concrete states of a program can be, in general, infinite, a common principle is *abstraction*. A good abstraction is, on the one hand, as coarse as possible —to keep the state space that must be explored small— and, on the other hand, precise enough to eliminate false alarms.

Tools for software model checking combine many different techniques, for example, counterexample-guided abstraction refinement (CEGAR) [21], predicate abstraction [31], bounded model checking (BMC) [16, 22], lazy abstraction [9, 34], k-induction [8, 27], and interpolation [23, 42]. A listing of the widely-used techniques, and which tools implement which technique, is given in the SV-COMP'17 report [4] in Table 4. In this work, we use a general-purpose bounded model checker (Cbmc), a sequential combination of approaches (CPA-Seq), a bounded model checker with incrementally increasing bounds (Esbmc-incr), and a k-induction based model checker (Esbmc-kInd).
Cbmc [22, 40] uses bit-precise BMC with MiniSat [26] as SAT-solver backend. BMC performs model checking with limited loop unrolling, i.e., loops are only unrolled up to a given bound. If no property violation can be found in the explored state space under this restriction, the program is assumed to be safe in general.
CPA-Seq [25] is based on CPAchecker that combines explicit-state model checking [13], k-induction [8], and predicate analysis with adjustable-block abstraction [12] sequentially. CPA-Seq uses the bit-precise SMT solver MathSAT5 [20].
Esbmc-incr [43] is a fork of Cbmc with an improved memory model. It uses an iterative scheme to increase its loop bounds, i.e., if no error is found in a program analysis using a certain loop bound, then the bound is increased. If no error is found after a set number of iterations, the program is assumed to be safe.
Esbmc-kInd [27] uses automatic k-induction to compute loop invariants in the context-bounded model checking of Esbmc. It performs the three phases of k-induction in parallel, which often yields a performance advantage.

2.3 Validation of Results

It is well-understood that when testers and model checkers produce test cases and error paths, respectively, sometimes the results contain false alarms. In order to avoid investing time on false results, test cases can be validated by reproducing a real crash [24, 41] and error paths can be evaluated by witness validation [7, 15]. A *violation witness* is an automaton that describes a set of paths through the

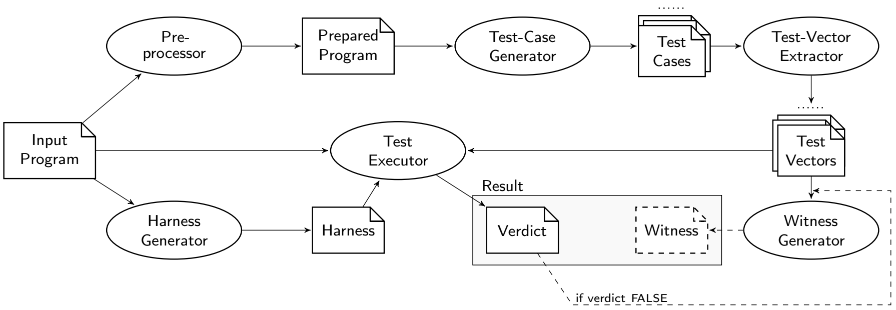


Fig. 1: Workflow of TBF

program that contain a specification violation. Each state transition contains a *source-code guard* that specifies the program-code locations at which the transition is allowed, and a *state-space guard* that constrains the set of possible program states after the transition. We considered four existing witness validators.

CPAchecker [7] uses predicate analysis with adjustable-block abstraction combined with explicit-state model checking for witness validation.

CPA-witness2test[7] creates a compilable test harness from a violation witness and checks whether the specification violation is reached through execution.

FShell-witness2test[8] also performs execution-based witness validation, but does not rely on any verification tool.

Ultimate Automizer [32] uses an automata-centric approach [33] to model checking for witness validation.

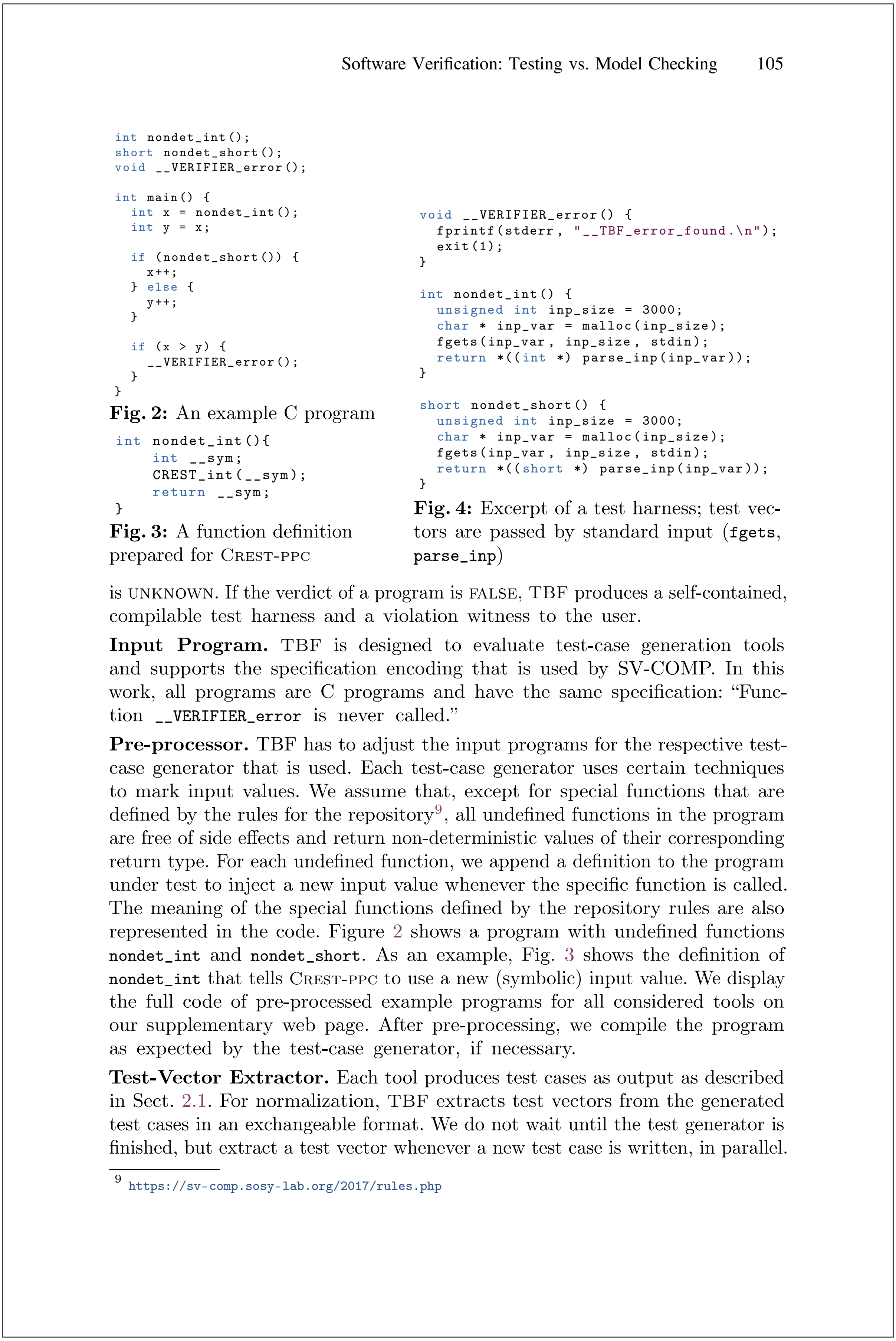
In this work, we evaluate the results from testers with TBF by considering for each test case, one by one, whether compiled with a test harness and the program, the execution violates the specification, and we evaluate the results of model checkers by validating the violation witness using four different witness validators. This way, we count bug reports only if they can be reproduced.

3 Framework for Test-Based Falsification

We designed a framework for test-based falsification (TBF) that makes it possible to uniformly use test-case generation tools. Figure 1 shows the architecture of this approach. Given an input program, TBF first pre-processes the program into the format that the test-case generator requires (‘prepared program’). This includes, e.g., adding function definitions for assigning new symbolic values and compiling the program in a certain way expected by the generator. The prepared program is then given to the test-case generator, which stores its output in its own, proprietary format (‘test cases’). These test cases are given to a test-vector extractor to extract the test vectors and store them in an exchangeable, uniform format (‘test vectors’). The harness generator produces a test harness for the input program, which is compiled and linked together with the input program and executed by the test executor. If the execution reports a specification violation, the verdict is FALSE. In all other cases, the verdict

[7] https://github.com/sosy-lab/cpachecker

[8] https://github.com/tautschnig/cprover-sv-comp/tree/test-gen/witness2test



```
int nondet_int();
short nondet_short();
void __VERIFIER_error();

int main() {
  int x = nondet_int();
  int y = x;

  if (nondet_short()) {
    x++;
  } else {
    y++;
  }

  if (x > y) {
    __VERIFIER_error();
  }
}
```

Fig. 2: An example C program

```
int nondet_int(){
    int __sym;
    CREST_int(__sym);
    return __sym;
}
```

Fig. 3: A function definition prepared for Crest-ppc

```
void __VERIFIER_error() {
  fprintf(stderr, "__TBF_error_found.\n");
  exit(1);
}

int nondet_int() {
  unsigned int inp_size = 3000;
  char * inp_var = malloc(inp_size);
  fgets(inp_var, inp_size, stdin);
  return *((int *) parse_inp(inp_var));
}

short nondet_short() {
  unsigned int inp_size = 3000;
  char * inp_var = malloc(inp_size);
  fgets(inp_var, inp_size, stdin);
  return *((short *) parse_inp(inp_var));
}
```

Fig. 4: Excerpt of a test harness; test vectors are passed by standard input (`fgets`, `parse_inp`)

is unknown. If the verdict of a program is false, TBF produces a self-contained, compilable test harness and a violation witness to the user.

Input Program. TBF is designed to evaluate test-case generation tools and supports the specification encoding that is used by SV-COMP. In this work, all programs are C programs and have the same specification: "Function `__VERIFIER_error` is never called."

Pre-processor. TBF has to adjust the input programs for the respective test-case generator that is used. Each test-case generator uses certain techniques to mark input values. We assume that, except for special functions that are defined by the rules for the repository[9], all undefined functions in the program are free of side effects and return non-deterministic values of their corresponding return type. For each undefined function, we append a definition to the program under test to inject a new input value whenever the specific function is called. The meaning of the special functions defined by the repository rules are also represented in the code. Figure 2 shows a program with undefined functions `nondet_int` and `nondet_short`. As an example, Fig. 3 shows the definition of `nondet_int` that tells Crest-ppc to use a new (symbolic) input value. We display the full code of pre-processed example programs for all considered tools on our supplementary web page. After pre-processing, we compile the program as expected by the test-case generator, if necessary.

Test-Vector Extractor. Each tool produces test cases as output as described in Sect. 2.1. For normalization, TBF extracts test vectors from the generated test cases in an exchangeable format. We do not wait until the test generator is finished, but extract a test vector whenever a new test case is written, in parallel.

[9] https://sv-comp.sosy-lab.org/2017/rules.php

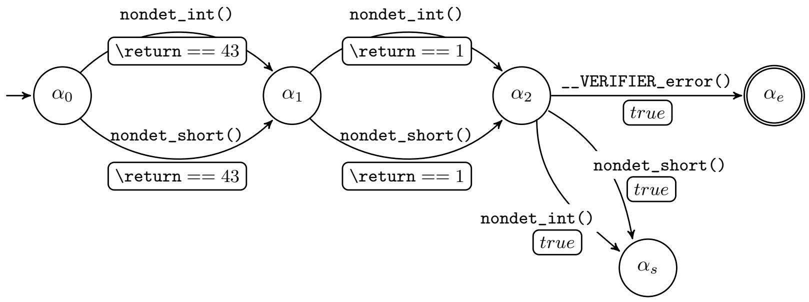


Fig. 5: Violation witness for test vector $\langle 43, 1\rangle$ and two non-deterministic methods

Harness Generator and Test Executor. We provide an effective and efficient way of checking whether a generated test vector represents a property violation: We create a test harness for the program under test that can feed an input value into the program for each call to a non-deterministic function. For performance reasons, it gets these input values from standard input. For each test vector extracted from the produced test cases, we execute the pre-compiled test harness with the vector as input and check whether a property violation occurs during execution. An example harness is shown in Fig. 4.

Witness Generation. A test vector $\langle \eta_1, \cdots, \eta_n\rangle$ can be represented by a violation witness that contains one initial state α_0, one accepting state α_e, one sink sate α_s, and, for each value η_i of the test vector, a state α_i with, for each non-deterministic function occurring in the program, a transition from α_{i-1} to α_i with the call to the corresponding function as source-code guard and η_i as return value for the corresponding function as state-space guard, i.e.: the transition can only be taken if the corresponding function is called, and, if the transition is taken, it is assumed that the return value of the corresponding function is η_i. From α_n, there is one transition to α_e for each occurring call to `__VERIFIER_error`, and one transition to α_s for each non-deterministic function in the program. Each such transition has the corresponding function call as source-code guard and no state-space guard. The transitions to sink state α_s make sure that no path is considered that may need an additional input value. While such a path may exist in the program, it can not be the path described by the test vector. Fig. 5 shows an example of such a witness. Each transition between states is labeled with the source-code guard (no box) and the state-space guard (boxed). The value '*true*' means that no state-space guard exists for that transition.

When validating the displayed violation witness, a validator explores the state-space until it encounters a call to `nondet_int` or `nondet_short`. Then, it is told to assume that the encountered function returns the concrete value 43, described by the special identifier `\return`. When it encounters one of the two functions for the second time, it is told to assume that the corresponding function returns the concrete value 1. After this, if it encounters a call to `__VERIFIER_error`, it confirms the violation witness. If it encounters a call to one of the two non-deterministic functions for the third time, it enters the sink state α_s, since our witnessed counterexample only contains two calls to non-deterministic functions.

4 Experimental Evaluation

We compare automatic test generators against automatic software model checkers regarding bug finding abilities in a large-scale experimental evaluation.

4.1 Experiment Setup

Programs under Test. To get a representative set of programs under test, we used all 5 693 verification tasks of the sv-benchmarks set[10] in revision `879e141f`[11] whose specification is that function `__VERIFIER_error` is not called. Of the 5 693 programs, 1 490 programs contain a known bug (at most one bug per program), i.e., there is a path through the program that ends in a call to `__VERIFIER_error`, and 4 203 programs are correct. The benchmark set is partitioned into categories. A description of the kinds of programs in the categories of an earlier version of the repository can be found in the literature (cf. [3], Sect. 4). For each category (e.g., 'Arrays'), the defining set of contained programs (`.set` file[12]), and a short characterization and the bit architecture of the contained programs (`.cfg` file[13]) can be found in the repository itself.

Availability. More details about the programs under test, generated test cases, generated witnesses, and other experimental data are available on the supplementary web page.[14]

Tools. We used the test generators and model checkers in the versions specified in Table 1. TBF[15] is implemented in Python `3.5.2` and available as open-source; we use TBF in version `0.1`. For Crest-ppc, we use a modified revision that supports long data types. For readability, we add superscripts T and M to the tool names for better visual identification of the testers and model checkers, respectively. We selected six testing tools that (i) support the language C, (ii) are freely available, (iii) cover a spectrum of different technologies, (iv) are available for 64-bit GNU/Linux, and (v) generate test cases for branch coverage or similar: AFL-fuzz, CPATiger, Crest-ppc, FShell, Klee, and PRtest. For the model checkers, we use the four most successful model checkers in category 'Falsification' of SV-COMP'17[16], i.e., Cbmc, CPA-Seq, Esbmc-incr, and Esbmc-kInd. To validate the results of violation witnesses, we use CPAchecker and Ultimate Automizer in the revision from SV-COMP'17, CPA-witness2test in revision `r24473` of the CPAchecker repository, and FShell-witness2test in revision `2a76669f` from branch `test-gen` in the Cprover repository[17].

Computing Resources. We performed all experiments on machines with an Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of memory, and a Ubuntu 16.04 operating system with kernel Linux 4.4.

[10] https://sv-comp.sosy-lab.org/2017/benchmarks.php

[11] https://github.com/sosy-lab/sv-benchmarks/tree/879e141f

[12] https://github.com/sosy-lab/sv-benchmarks/blob/879e141f/c/ReachSafety-Arrays.set

[13] https://github.com/sosy-lab/sv-benchmarks/blob/879e141f/c/ReachSafety-Arrays.cfg

[14] https://www.sosy-lab.org/research/test-study/

[15] https://github.com/sosy-lab/tbf

[16] https://sv-comp.sosy-lab.org/2017/results/

[17] https://github.com/tautschnig/cprover-sv-comp

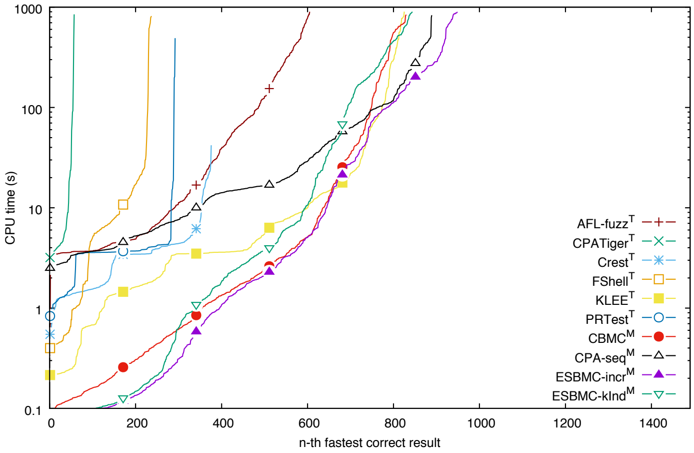


Fig. 6: Quantile plots for the different tools for finding bugs in programs

We limited each benchmark run to 2 processing units, 15 GB of memory, and 15 min of CPU time. All CPU times are reported with two significant digits.

4.2 Experimental Results

Now we report the results of our experimental study. For each of the 1 490 programs that contain a known bug, we applied all testers and model checkers in order to find the bug. For the testers, a bug is found if one of the generated test cases executes the undesired function call. For the model checkers, a bug is found if the tools returns answer FALSE together with a violation witness.

Qualitative Overview. We illustrate the overall picture using the quantile plot in Fig. 6. For each data point (x, y) on a graph, the quantile plot shows that x bugs can be correctly identified using at most y seconds of CPU time. The x-position of the right-most data point for a tool indicates the total number of bugs the tool was able to identify. In summary, each model checker finds more bugs than the best tester, while the best tester (KLEE$^{\text{T}}$) closely follows the weakest model checker (CBMC$^{\text{M}}$).

The area below the graph is proportional to the overall consumed CPU time for successfully solved problems. The visualization makes it easy to see, e.g., by looking at the 400 fastest solved problems, that most testers time out while most model checkers use only a fraction of their available CPU time. In summary, the ratio of returned results by invested resources is much better for the model checkers.

Quantitative Overview. Next, we look at the numerical details as shown in Table 2. The columns are partitioned into four parts: the table lists (i) the category/row label together with the number of programs (maximal number of found bugs), (ii) the number of found bugs for the six testers, (iii) the number of found bugs for the four model checkers, and (iv) the union of the results for testers, model checkers, and overall. In the two parts for the testers and model checkers, we highlight the best result in bold (if equal, the fastest result is highlighted). The rows are partitioned into three parts: the table shows first

Table 2: Results for testers and model checkers on programs with a bug

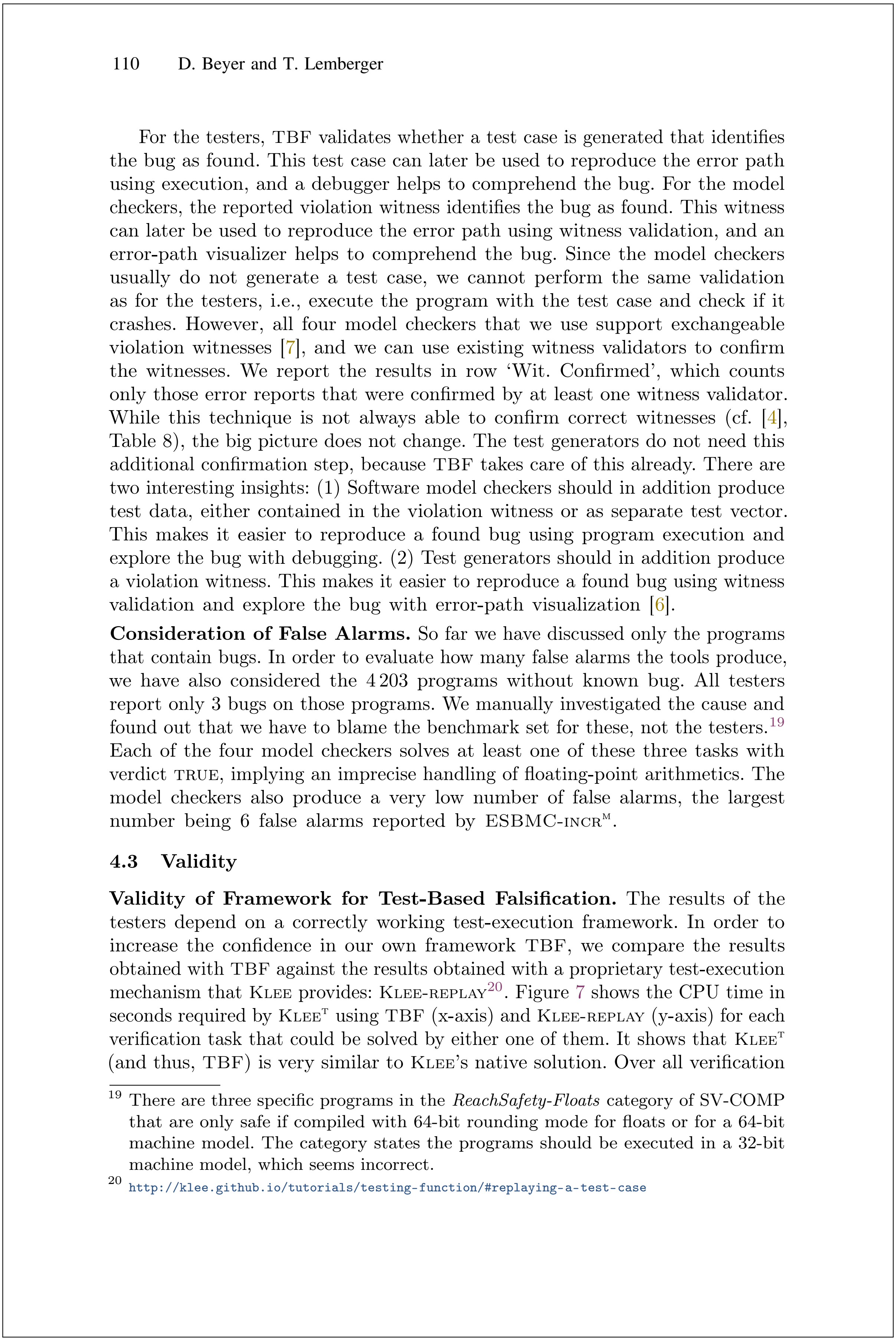
	No. Programs	AFL-fuzz$^{\mathrm{T}}$	CPATiger$^{\mathrm{T}}$	Crest-ppc$^{\mathrm{T}}$	FShell$^{\mathrm{T}}$	Klee$^{\mathrm{T}}$	PRtest$^{\mathrm{T}}$	CBMC$^{\mathrm{M}}$	CPA-seq$^{\mathrm{M}}$	ESBMC-incr$^{\mathrm{M}}$	ESBMC-kInd$^{\mathrm{M}}$	Union Testers	Union MC	Union All
Arrays	81	**26**	0	20	4	22	25	**6**	3	6	4	31	13	33
BitVectors	24	**11**	5	7	5	11	10	12	12	12	**12**	14	17	19
ControlFlow	42	15	0	11	3	**20**	3	**41**	23	36	35	21	42	42
ECA	413	234	0	51	0	**260**	0	143	**257**	221	169	286	42	338
Floats	31	**11**	2	2	4	2	11	**31**	29	17	13	13	31	31
Heap	66	46	22	16	13	**48**	32	**64**	31	62	58	48	66	66
Loops	46	**45**	27	29	5	40	33	**42**	36	42	38	41	38	43
ProductLines	265	169	1	204	156	**255**	144	263	**265**	265	263	265	265	265
Recursive	45	44	0	35	22	**45**	31	**42**	41	40	40	45	43	45
Sequentialized	170	4	0	1	24	**123**	3	**135**	122	135	134	123	141	147
LDV	307	0	0	0	0	0	0	51	70	**113**	78	0	147	147
Total Found	1 490	605	57	376	236	**826**	292	830	889	**949**	844	887	1 092	1 176
Compilable	1 115	605	57	376	236	**826**	292	779	819	**830**	761	887	930	1 014
Wit. Confirmed	1 490							761	**857**	705	634	887	979	1 068
Median CPU Time (s)		11	4.5	**3.4**	6.2	3.6	3.6	**1.4**	15	1.9	2.3			
Average CPU Time (s)		82	38	**4.1**	27	33	6.7	**46**	51	61	69			

the results for each of the 11 categories of the programs under test, second the results for all categories together, and third the CPU times required.

The row 'Total Found' shows that the best tester (Klee$^{\mathrm{T}}$) is able to find 826 bugs, while all model checkers find more, with the best model checker (ESBMC-incr$^{\mathrm{M}}$) finding 15 % more bugs (949) than the best tester. An interesting observation is that the different tools have different strengths and weaknesses: column 'Union Testers' shows that applying all testers together increases the amount of solved tasks considerably. This is made possible using our unifying framework TBF, which abstracts from the differences in input and output of the various tools and lets us use all testers in a common work flow. The same holds for the model checkers: the combination of all approaches significantly increases the number of solved problems (column 'Union MC'). The combination of testers and model checkers (column 'Union All') in a bug-finding workflow can further improve the results significantly, i.e., there are program bugs that one technique can find but not the other, and vice versa.

While it is usually considered an advantage that model checkers can be applied to incomplete programs that are not yet fully defined (as expected by static-analysis tools), testers obviously cannot be applied to such programs (as they are dynamic-analysis tools). This issue applies in particular to the category 'LDV' of device drivers, which contain non-deterministic models of the operating-system environment. This kind of programs is important because it is successfully used to find bugs in systems code [18] [47], but in order to provide a comparison without the influence of this issue, we also report the results restricted to those programs that are compilable (row 'Compilable').

[18] http://linuxtesting.org/results/ldv



For the testers, TBF validates whether a test case is generated that identifies the bug as found. This test case can later be used to reproduce the error path using execution, and a debugger helps to comprehend the bug. For the model checkers, the reported violation witness identifies the bug as found. This witness can later be used to reproduce the error path using witness validation, and an error-path visualizer helps to comprehend the bug. Since the model checkers usually do not generate a test case, we cannot perform the same validation as for the testers, i.e., execute the program with the test case and check if it crashes. However, all four model checkers that we use support exchangeable violation witnesses [7], and we can use existing witness validators to confirm the witnesses. We report the results in row 'Wit. Confirmed', which counts only those error reports that were confirmed by at least one witness validator. While this technique is not always able to confirm correct witnesses (cf. [4], Table 8), the big picture does not change. The test generators do not need this additional confirmation step, because TBF takes care of this already. There are two interesting insights: (1) Software model checkers should in addition produce test data, either contained in the violation witness or as separate test vector. This makes it easier to reproduce a found bug using program execution and explore the bug with debugging. (2) Test generators should in addition produce a violation witness. This makes it easier to reproduce a found bug using witness validation and explore the bug with error-path visualization [6].

Consideration of False Alarms. So far we have discussed only the programs that contain bugs. In order to evaluate how many false alarms the tools produce, we have also considered the 4 203 programs without known bug. All testers report only 3 bugs on those programs. We manually investigated the cause and found out that we have to blame the benchmark set for these, not the testers.[19] Each of the four model checkers solves at least one of these three tasks with verdict TRUE, implying an imprecise handling of floating-point arithmetics. The model checkers also produce a very low number of false alarms, the largest number being 6 false alarms reported by ESBMC-INCR$^{\mathrm{M}}$.

4.3 Validity

Validity of Framework for Test-Based Falsification. The results of the testers depend on a correctly working test-execution framework. In order to increase the confidence in our own framework TBF, we compare the results obtained with TBF against the results obtained with a proprietary test-execution mechanism that Klee provides: Klee-replay[20]. Figure 7 shows the CPU time in seconds required by Klee$^{\mathrm{T}}$ using TBF (x-axis) and Klee-replay (y-axis) for each verification task that could be solved by either one of them. It shows that Klee$^{\mathrm{T}}$ (and thus, TBF) is very similar to Klee's native solution. Over all verification

[19] There are three specific programs in the *ReachSafety-Floats* category of SV-COMP that are only safe if compiled with 64-bit rounding mode for floats or for a 64-bit machine model. The category states the programs should be executed in a 32-bit machine model, which seems incorrect.

[20] http://klee.github.io/tutorials/testing-function/#replaying-a-test-case

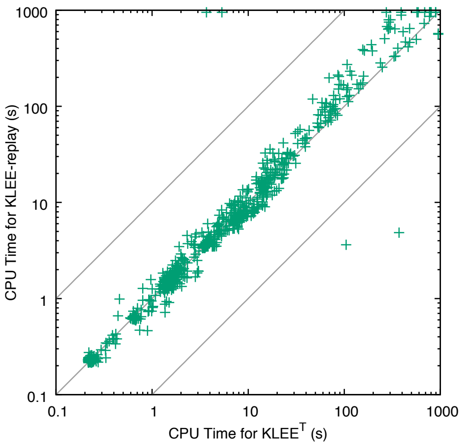


Fig. 7: CPU time required by KLEE^{T} and KLEE-REPLAY to solve tasks

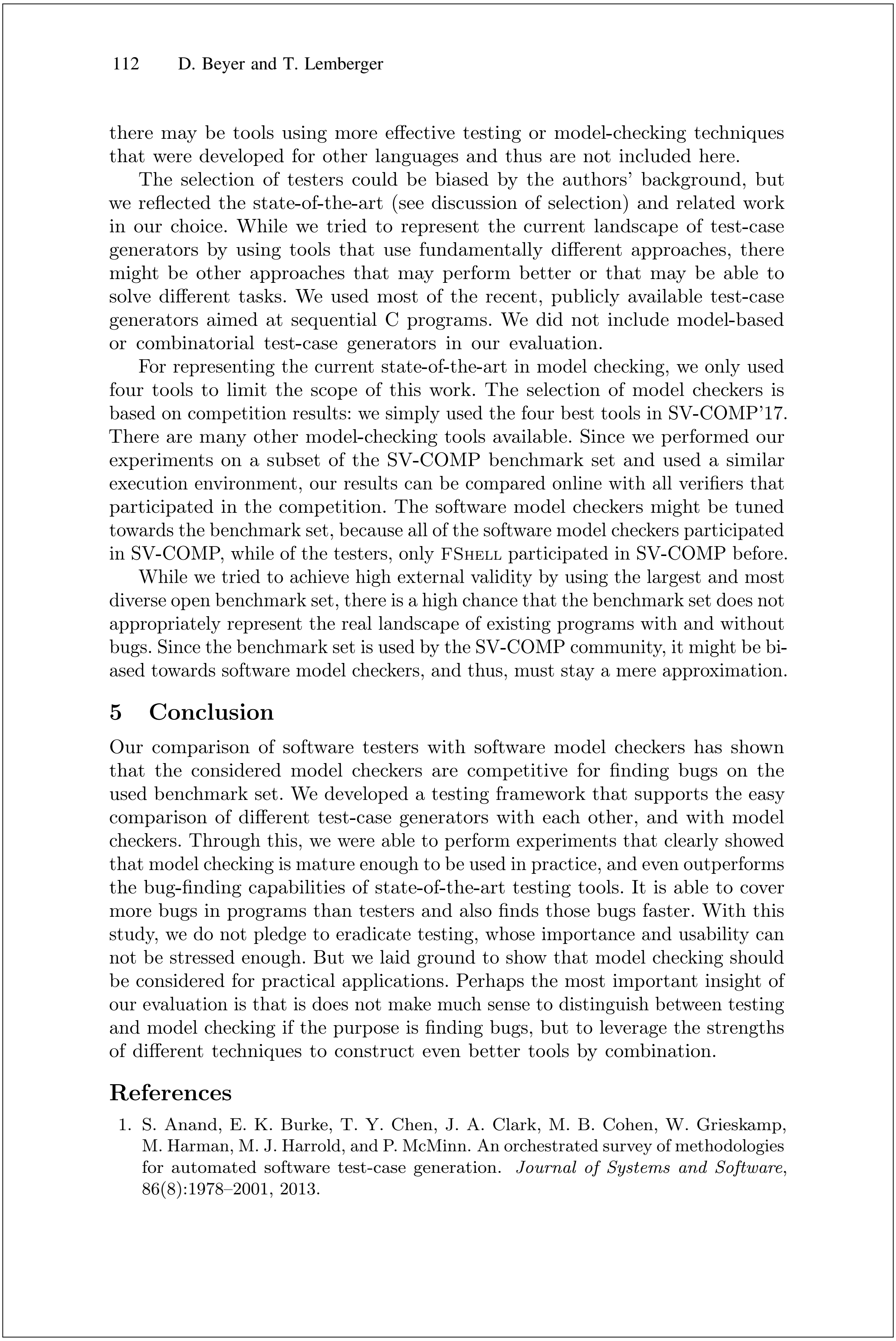
tasks, KLEE^{T} is able to find bugs in 826 tasks, while KLEE-REPLAY is able to find bugs in 821 tasks. There are 15 tasks that KLEE-REPLAY can not solve, while KLEE^{T} can, and 10 tasks that KLEE-REPLAY can solve, while KLEE^{T} can not.

For KLEE^{T}, one unsolved task is due to missing support of a corner case for the conversion of KLEE's internal representation of numbers to a test vector. The remaining difference is due to an improper machine model: for KLEE-REPLAY, we only had 64-bit libraries available, while most tasks of SV-COMP are intended to be analyzed using a 32-bit architecture. This only results in a single false result, but interprets some of the inputs generated for 32-bit programs differently, thus reaching different parts of the program in a few cases. This also explains the few outliers in Fig. 7. The two implementations both need a median of 0.43 s of CPU time to find a bug in a task. This shows that our implementation is similarly effective and efficient to KLEE's own, tailored test-execution mechanism.

Other Threats to Internal Validity. We used the state-of-the-art benchmarking tool BENCHEXEC [14] to run every execution in an isolated container with dedicated resources, making our results as reliable as possible. Our experimental results for the considered model checkers are very close to the results of SV-COMP'17[21], indicating their accuracy. Our framework TBF is a prototype and may contain bugs that degrade the real performance of test-based falsification. Probably more tasks could be solved if more time was invested in improving this approach, but we tried to keep our approach as simple as possible to influence the results as less as possible.

Threats to External Validity. There are several threats to external validity. All tools that we evaluated are aimed at analyzing C programs. It might be the case that testing research is focused on other languages, such as C++ or Java. Other languages may contain other quirks than C that make certain approaches to test-case generation and model checking more or less successful. In addition,

[21] https://sv-comp.sosy-lab.org/2017/results/



there may be tools using more effective testing or model-checking techniques that were developed for other languages and thus are not included here.

The selection of testers could be biased by the authors' background, but we reflected the state-of-the-art (see discussion of selection) and related work in our choice. While we tried to represent the current landscape of test-case generators by using tools that use fundamentally different approaches, there might be other approaches that may perform better or that may be able to solve different tasks. We used most of the recent, publicly available test-case generators aimed at sequential C programs. We did not include model-based or combinatorial test-case generators in our evaluation.

For representing the current state-of-the-art in model checking, we only used four tools to limit the scope of this work. The selection of model checkers is based on competition results: we simply used the four best tools in SV-COMP'17. There are many other model-checking tools available. Since we performed our experiments on a subset of the SV-COMP benchmark set and used a similar execution environment, our results can be compared online with all verifiers that participated in the competition. The software model checkers might be tuned towards the benchmark set, because all of the software model checkers participated in SV-COMP, while of the testers, only FShell participated in SV-COMP before.

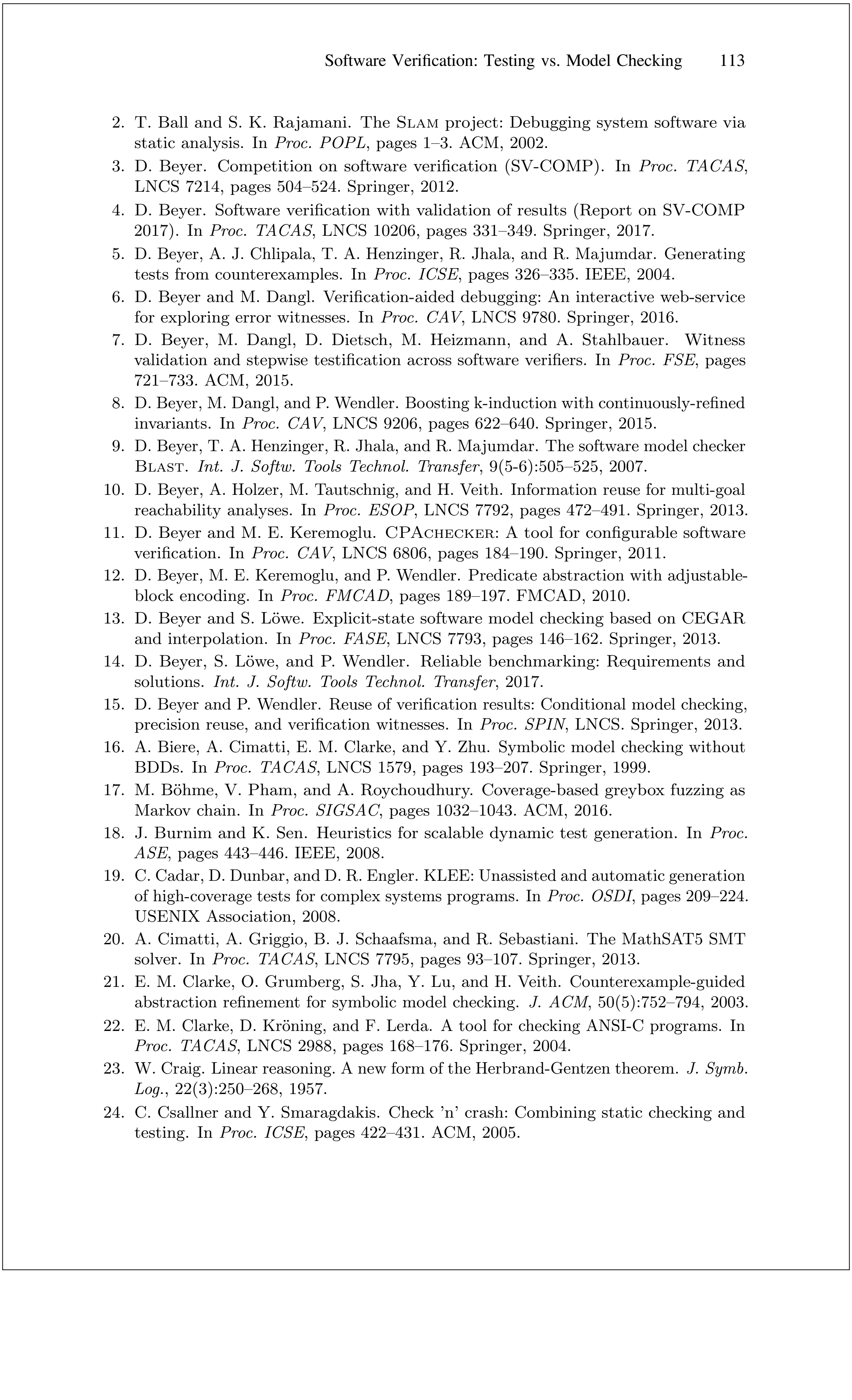
While we tried to achieve high external validity by using the largest and most diverse open benchmark set, there is a high chance that the benchmark set does not appropriately represent the real landscape of existing programs with and without bugs. Since the benchmark set is used by the SV-COMP community, it might be biased towards software model checkers, and thus, must stay a mere approximation.

5 Conclusion

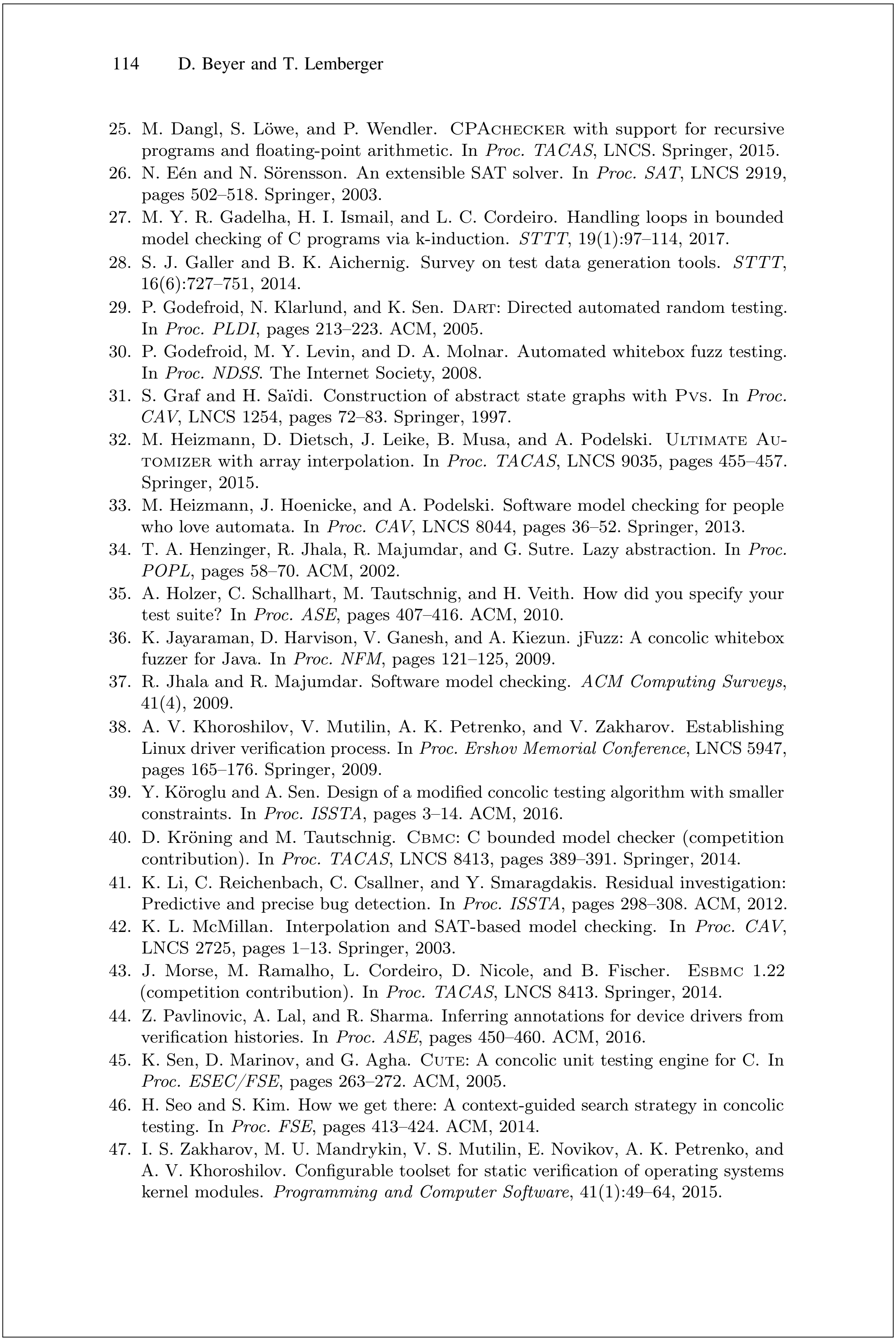
Our comparison of software testers with software model checkers has shown that the considered model checkers are competitive for finding bugs on the used benchmark set. We developed a testing framework that supports the easy comparison of different test-case generators with each other, and with model checkers. Through this, we were able to perform experiments that clearly showed that model checking is mature enough to be used in practice, and even outperforms the bug-finding capabilities of state-of-the-art testing tools. It is able to cover more bugs in programs than testers and also finds those bugs faster. With this study, we do not pledge to eradicate testing, whose importance and usability can not be stressed enough. But we laid ground to show that model checking should be considered for practical applications. Perhaps the most important insight of our evaluation is that is does not make much sense to distinguish between testing and model checking if the purpose is finding bugs, but to leverage the strengths of different techniques to construct even better tools by combination.

References

1. S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test-case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.



2. T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
3. D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.
4. D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017.
5. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. ICSE*, pages 326–335. IEEE, 2004.
6. D. Beyer and M. Dangl. Verification-aided debugging: An interactive web-service for exploring error witnesses. In *Proc. CAV*, LNCS 9780. Springer, 2016.
7. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proc. FSE*, pages 721–733. ACM, 2015.
8. D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. CAV*, LNCS 9206, pages 622–640. Springer, 2015.
9. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
10. D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. Information reuse for multi-goal reachability analyses. In *Proc. ESOP*, LNCS 7792, pages 472–491. Springer, 2013.
11. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
12. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
13. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.
14. D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer*, 2017.
15. D. Beyer and P. Wendler. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proc. SPIN*, LNCS. Springer, 2013.
16. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, LNCS 1579, pages 193–207. Springer, 1999.
17. M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proc. SIGSAC*, pages 1032–1043. ACM, 2016.
18. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446. IEEE, 2008.
19. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224. USENIX Association, 2008.
20. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Proc. TACAS*, LNCS 7795, pages 93–107. Springer, 2013.
21. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
22. E. M. Clarke, D. Kröning, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, LNCS 2988, pages 168–176. Springer, 2004.
23. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.
24. C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. ICSE*, pages 422–431. ACM, 2005.



25. M. Dangl, S. Löwe, and P. Wendler. CPAchecker with support for recursive programs and floating-point arithmetic. In *Proc. TACAS*, LNCS. Springer, 2015.
26. N. Eén and N. Sörensson. An extensible SAT solver. In *Proc. SAT*, LNCS 2919, pages 502–518. Springer, 2003.
27. M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro. Handling loops in bounded model checking of C programs via k-induction. *STTT*, 19(1):97–114, 2017.
28. S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *STTT*, 16(6):727–751, 2014.
29. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.
30. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*. The Internet Society, 2008.
31. S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.
32. M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate Automizer with array interpolation. In *Proc. TACAS*, LNCS 9035, pages 455–457. Springer, 2015.
33. M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *Proc. CAV*, LNCS 8044, pages 36–52. Springer, 2013.
34. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
35. A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. How did you specify your test suite? In *Proc. ASE*, pages 407–416. ACM, 2010.
36. K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *Proc. NFM*, pages 121–125, 2009.
37. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.
38. A. V. Khoroshilov, V. Mutilin, A. K. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In *Proc. Ershov Memorial Conference*, LNCS 5947, pages 165–176. Springer, 2009.
39. Y. Köroglu and A. Sen. Design of a modified concolic testing algorithm with smaller constraints. In *Proc. ISSTA*, pages 3–14. ACM, 2016.
40. D. Kröning and M. Tautschnig. Cbmc: C bounded model checker (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 389–391. Springer, 2014.
41. K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. Residual investigation: Predictive and precise bug detection. In *Proc. ISSTA*, pages 298–308. ACM, 2012.
42. K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.
43. J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer. Esbmc 1.22 (competition contribution). In *Proc. TACAS*, LNCS 8413. Springer, 2014.
44. Z. Pavlinovic, A. Lal, and R. Sharma. Inferring annotations for device drivers from verification histories. In *Proc. ASE*, pages 450–460. ACM, 2016.
45. K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272. ACM, 2005.
46. H. Seo and S. Kim. How we get there: A context-guided search strategy in concolic testing. In *Proc. FSE*, pages 413–424. ACM, 2014.
47. I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. *Programming and Computer Software*, 41(1):49–64, 2015.

International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-020-00568-x

COMPETITIONS AND CHALLENGES

Special Issue: TestComp 2019

Plain random test generation with PRTest

Thomas Lemberger[1]

© The Author(s) 2020

Abstract
Automatic test-suite generation tools are often complex and their behavior is not predictable. To provide a minimum baseline that test-suite generators should be able to surpass, we present PRTEST, a random black-box test-suite generator for C programs: To create a test, PRTEST natively executes the program under test and creates a new, random test value whenever an input value is required. After execution, PRTEST checks whether any new program branches were covered and, if this is the case, the created test is added to the test suite. This way, tests are rapidly created either until a crash is found, or until the user aborts the creation. While this naive mechanism is not competitive with more sophisticated, state-of-the-art test-suite generation tools, it is able to provide a good baseline for Test-Comp and a fast alternative for automatic test-suite generation for programs with simple control flow. PRTEST is publicly available and open source.

Keywords Random testing · Software engineering · Software testing · Software verification · Test-Comp

1 Introduction

Automatic test-suite generation is a highly active field of research and many successful tools exist to this date. Unfortunately, most of these tools are based on sophisticated algorithms and thus, both their code and their behavior can be hard to understand for non-experts. In addition, these tools and their improvements are usually only compared to each other, but no naive baseline exists. We present PRTEST, a plain random test-suite generator that provides a solution for both issues. PRTEST is designed to be simple: its full test-suite generation logic consists of 125 lines of code, and it uses no heuristics or sophisticated algorithms. Instead, PRTEST provides random input generation [2]: It repeatedly executes the program under test with random inputs and stores the input values of an execution as a test if the execution increased the overall coverage. Thanks to its pure randomness and native execution of the program under test its behavior is easy to understand and it can be used as a lower baseline for Test-Comp.

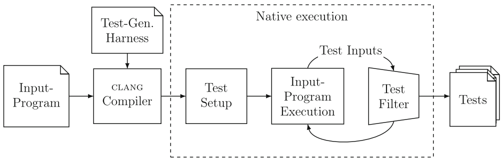


Fig. 1 Workflow of PRTEST

2 Test-suite generation approach

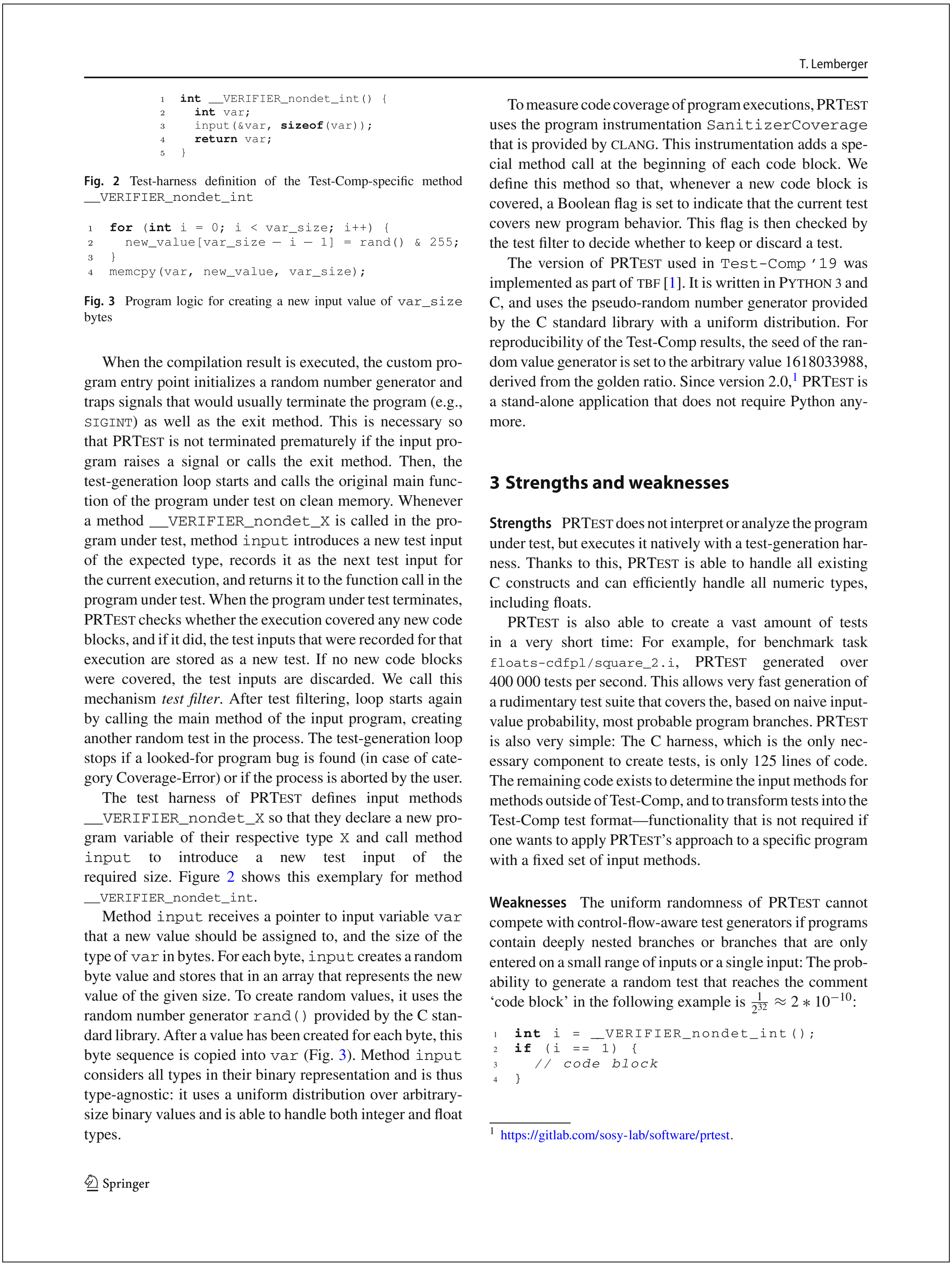
Figure 1 shows the workflow of PRTEST. PRTEST consists of two steps: (1) it compiles the input program against a test harness, and (2) it natively executes the compilation result. This execution consists of the test setup and a test-generation loop.

First, PRTEST uses the CLANG compiler to compile the program under test against a C harness that provides the full test-suite generation logic ('Test Gen. Harness' in Fig. 1). The harness provides: (1) a custom program entry point for test-generation setup, (2) definitions for the Test-Comp-specific input methods `__VERIFIER_nondet_X` (where `X` is any primitive C type; e.g., `__VERIFIER_nondet_int`), (3) a method `input` that creates new test inputs, and (4) CLANG-specific methods that allow PRTEST to track program coverage during runtime.

✉ Thomas Lemberger
thomas.lemberger@sosy.ifi.lmu.de

[1] LMU Munich, Munich, Germany

Published online: 06 July 2020



```
int __VERIFIER_nondet_int() {
  int var;
  input(&var, sizeof(var));
  return var;
}
```

Fig. 2 Test-harness definition of the Test-Comp-specific method `__VERIFIER_nondet_int`

```
for (int i = 0; i < var_size; i++) {
  new_value[var_size - i - 1] = rand() & 255;
}
memcpy(var, new_value, var_size);
```

Fig. 3 Program logic for creating a new input value of `var_size` bytes

When the compilation result is executed, the custom program entry point initializes a random number generator and traps signals that would usually terminate the program (e.g., `SIGINT`) as well as the exit method. This is necessary so that PRTEST is not terminated prematurely if the input program raises a signal or calls the exit method. Then, the test-generation loop starts and calls the original main function of the program under test on clean memory. Whenever a method `__VERIFIER_nondet_X` is called in the program under test, method `input` introduces a new test input of the expected type, records it as the next test input for the current execution, and returns it to the function call in the program under test. When the program under test terminates, PRTEST checks whether the execution covered any new code blocks, and if it did, the test inputs that were recorded for that execution are stored as a new test. If no new code blocks were covered, the test inputs are discarded. We call this mechanism *test filter*. After test filtering, loop starts again by calling the main method of the input program, creating another random test in the process. The test-generation loop stops if a looked-for program bug is found (in case of category Coverage-Error) or if the process is aborted by the user.

The test harness of PRTEST defines input methods `__VERIFIER_nondet_X` so that they declare a new program variable of their respective type `X` and call method `input` to introduce a new test input of the required size. Figure 2 shows this exemplary for method `__VERIFIER_nondet_int`.

Method `input` receives a pointer to input variable `var` that a new value should be assigned to, and the size of the type of `var` in bytes. For each byte, `input` creates a random byte value and stores that in an array that represents the new value of the given size. To create random values, it uses the random number generator `rand()` provided by the C standard library. After a value has been created for each byte, this byte sequence is copied into `var` (Fig. 3). Method `input` considers all types in their binary representation and is thus type-agnostic: it uses a uniform distribution over arbitrary-size binary values and is able to handle both integer and float types.

To measure code coverage of program executions, PRTEST uses the program instrumentation `SanitizerCoverage` that is provided by CLANG. This instrumentation adds a special method call at the beginning of each code block. We define this method so that, whenever a new code block is covered, a Boolean flag is set to indicate that the current test covers new program behavior. This flag is then checked by the test filter to decide whether to keep or discard a test.

The version of PRTEST used in `Test-Comp '19` was implemented as part of TBF [1]. It is written in PYTHON 3 and C, and uses the pseudo-random number generator provided by the C standard library with a uniform distribution. For reproducibility of the Test-Comp results, the seed of the random value generator is set to the arbitrary value 1618033988, derived from the golden ratio. Since version 2.0,[1] PRTEST is a stand-alone application that does not require Python anymore.

3 Strengths and weaknesses

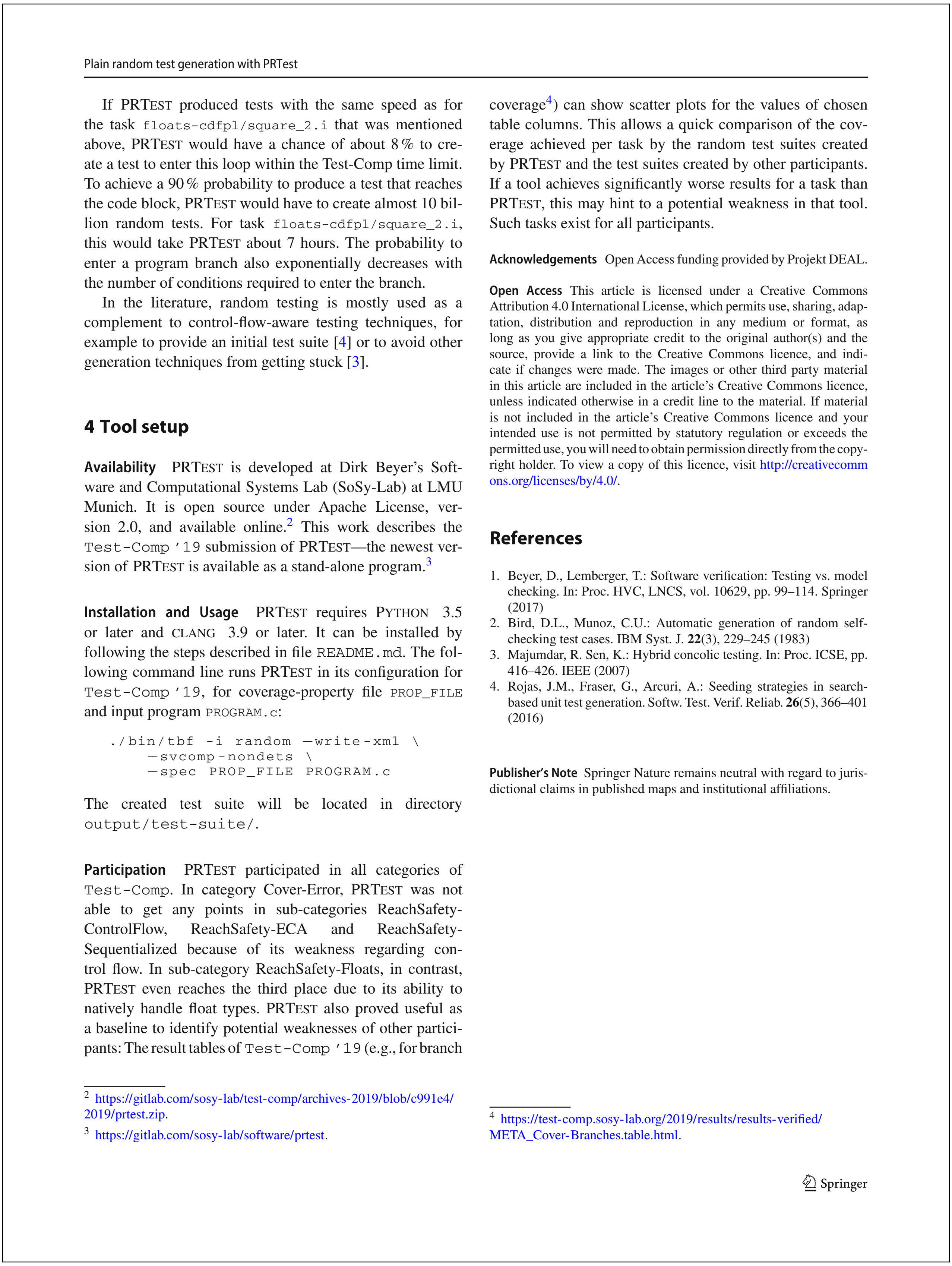
Strengths PRTEST does not interpret or analyze the program under test, but executes it natively with a test-generation harness. Thanks to this, PRTEST is able to handle all existing C constructs and can efficiently handle all numeric types, including floats.

PRTEST is also able to create a vast amount of tests in a very short time: For example, for benchmark task `floats-cdfpl/square_2.i`, PRTEST generated over 400 000 tests per second. This allows very fast generation of a rudimentary test suite that covers the, based on naive input-value probability, most probable program branches. PRTEST is also very simple: The C harness, which is the only necessary component to create tests, is only 125 lines of code. The remaining code exists to determine the input methods for methods outside of Test-Comp, and to transform tests into the Test-Comp test format—functionality that is not required if one wants to apply PRTEST's approach to a specific program with a fixed set of input methods.

Weaknesses The uniform randomness of PRTEST cannot compete with control-flow-aware test generators if programs contain deeply nested branches or branches that are only entered on a small range of inputs or a single input: The probability to generate a random test that reaches the comment 'code block' in the following example is $\frac{1}{2^{32}} \approx 2 * 10^{-10}$:

```
int i = __VERIFIER_nondet_int();
if (i == 1) {
  // code block
}
```

[1] https://gitlab.com/sosy-lab/software/prtest.



If PRTEST produced tests with the same speed as for the task `floats-cdfpl/square_2.i` that was mentioned above, PRTEST would have a chance of about 8 % to create a test to enter this loop within the Test-Comp time limit. To achieve a 90 % probability to produce a test that reaches the code block, PRTEST would have to create almost 10 billion random tests. For task `floats-cdfpl/square_2.i`, this would take PRTEST about 7 hours. The probability to enter a program branch also exponentially decreases with the number of conditions required to enter the branch.

In the literature, random testing is mostly used as a complement to control-flow-aware testing techniques, for example to provide an initial test suite [4] or to avoid other generation techniques from getting stuck [3].

4 Tool setup

Availability PRTEST is developed at Dirk Beyer's Software and Computational Systems Lab (SoSy-Lab) at LMU Munich. It is open source under Apache License, version 2.0, and available online.[2] This work describes the `Test-Comp '19` submission of PRTEST—the newest version of PRTEST is available as a stand-alone program.[3]

Installation and Usage PRTEST requires PYTHON 3.5 or later and CLANG 3.9 or later. It can be installed by following the steps described in file `README.md`. The following command line runs PRTEST in its configuration for `Test-Comp '19`, for coverage-property file `PROP_FILE` and input program `PROGRAM.c`:

```
./bin/tbf -i random —write-xml \
       —svcomp-nondets \
       —spec PROP_FILE PROGRAM.c
```

The created test suite will be located in directory `output/test-suite/`.

Participation PRTEST participated in all categories of `Test-Comp`. In category Cover-Error, PRTEST was not able to get any points in sub-categories ReachSafety-ControlFlow, ReachSafety-ECA and ReachSafety-Sequentialized because of its weakness regarding control flow. In sub-category ReachSafety-Floats, in contrast, PRTEST even reaches the third place due to its ability to natively handle float types. PRTEST also proved useful as a baseline to identify potential weaknesses of other participants: The result tables of `Test-Comp '19` (e.g., for branch coverage[4]) can show scatter plots for the values of chosen table columns. This allows a quick comparison of the coverage achieved per task by the random test suites created by PRTEST and the test suites created by other participants. If a tool achieves significantly worse results for a task than PRTEST, this may hint to a potential weakness in that tool. Such tasks exist for all participants.

Acknowledgements Open Access funding provided by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC, LNCS, vol. 10629, pp. 99–114. Springer (2017)
2. Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM Syst. J. **22**(3), 229–245 (1983)
3. Majumdar, R. Sen, K.: Hybrid concolic testing. In: Proc. ICSE, pp. 416–426. IEEE (2007)
4. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Softw. Test. Verif. Reliab. **26**(5), 366–401 (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

[2] https://gitlab.com/sosy-lab/test-comp/archives-2019/blob/c991e4/2019/prtest.zip.

[3] https://gitlab.com/sosy-lab/software/prtest.

[4] https://test-comp.sosy-lab.org/2019/results/results-verified/META_Cover-Branches.table.html.

2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)

TestCov: Robust Test-Suite Execution and Coverage Measurement

Dirk Beyer
LMU Munich, Germany

Thomas Lemberger
LMU Munich, Germany

***Abstract*—We present TestCov, a tool for robust test-suite execution and test-coverage measurement on C programs. TestCov executes program tests in isolated containers to ensure system integrity and reliable resource control. The tool provides coverage statistics per test and for the whole test suite. TestCov uses the simple, XML-based exchange format for test-suite specifications that was established as standard by Test-Comp. TestCov has been successfully used in Test-Comp '19 to execute almost 9 million tests on 1 720 different programs. The source code of TestCov is released under the open-source license Apache 2.0 and available at https://gitlab.com/sosy-lab/software/test-suite-validator. A full artifact, including a demonstration video, is available at https://doi.org/10.5281/zenodo.3418726.**

***Index Terms*—Test Execution, Coverage, Test-Suite Reduction**

I. Introduction

Modern test-case generators are able to generate system tests that reveal bugs in programs like never before, but executing these tests may lead to system failures, modifications, information leakage, or resource exhaustion. Because of this, program tests are often executed in virtual machines or containers (e.g., Docker). TestCov provides a lightweight solution to this: it uses an overlay file system and Linux control groups to protect the file system from modifications and to prevent unexpected resource usage during test execution, based on the existing benchmarking tool BenchExec [4]. Compared to other containerization technology, BenchExec does not require the installation of any additional software or superuser privileges during usage because it solely relies on features built into the Linux kernel. TestCov provides coverage statistics for line, branch, and condition coverage per test and for the whole test suite, and creates plots to visualize the measured data.

TestCov has been used in the First International Competition on Software Testing (Test-Comp '19) [1] to validate the test suites created by all 9 participants. TestCov uses the simple, XML-based exchange format for test-suite specification that was established as a standard by Test-Comp. All 9 participants support the exchange format. In the past, test-case generators used proprietary formats to output their generated tests, which led to two problems: Test suites can, depending on the format, often only be executed using auxiliary programs or not at all, and test suites generated by different test-case generators can not be directly compared or combined. The XML-based exchange format solves these issues.

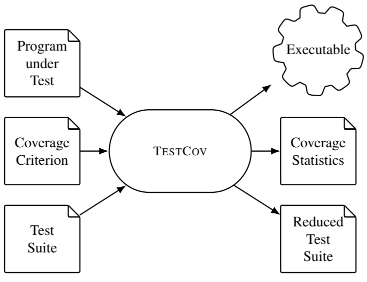


Fig. 1: Inputs and outputs of TestCov

Availability. TestCov is publicly available via GitLab [1] and as an archived package [5].

Related Work. TestCov aims at unifying and executing test suites that are created by test-case generators [7], [13], [6], [2], [9], [10], [8] for C programs. Klee [7] provides a *replay library* that can be used to create a test harness from the program under test with which it is possible to execute individual tests in the proprietary test-case format of Klee. The test cases created by AFL-fuzz[2] can be directly fed to a program. None of the existing test executors supports tests that are created by other test-case generators, nor the execution of a full test suite.

TestCov is based on BenchExec; other tools for containerization are Docker [3], LXC [4], and Snap [5]. Other projects related to the isolation and robust execution of software bugs are BugZoo [6] and the ManyBugs and IntroClass benchmarks [12].

II. Architecture of TestCov

Figure 1 shows the inputs and outputs of TestCov. TestCov gets as input the C program under test, the coverage criterion to check against, and the test suite, and creates an executable program that can be used to feed tests to the program under test, coverage statistics about the test suite, and a reduced test suite that achieves the same coverage (with respect to the coverage criterion) as the original test suite.

Supported in part by DFG grant BE 1761/7-1.

[1] https://gitlab.com/sosy-lab/software/test-suite-validator
[2] http://lcamtuf.coredump.cx/afl/
[3] https://www.docker.com/
[4] https://linuxcontainers.org/
[5] https://snapcraft.io/
[6] https://github.com/squaresLab/BugZoo

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00105

```
<?xml version="1.0"?>
<!DOCTYPE test-metadata PUBLIC [...]>
<test-metadata>
  <sourcecodelang>C</sourcecodelang>
  <producer>testcov v3.0</producer>
  <specification>CHECK( FQL(cover EDGES(@CONDITIONEDGE)) )</specification>
  <programfile>example.c</programfile>
  <programhash>eeecda9cbf27c43c9017fa00dd900c19a5ec18d46303f59a6e0357db78c33849</programhash>
  <entryfunction>main</entryfunction>
  <architecture>32bit</architecture>
  <inputtestsuitefile>original-suite.zip</inputtestsuitefile>
  <inputtestsuitehash>11911d658dcfbf8501390bf0faa96eb193b11bb1</inputtestsuitehash>
  <creationtime>2019-06-19T14:17:34Z</creationtime>
</test-metadata>
```

Fig. 2: Example metadata file of a test suite

```
<?xml version="1.0"?>
<!DOCTYPE testcase PUBLIC [...]>
<testcase>
  <input>'b'</input>
  <input>10</input>
  <input>0x0f</input>
</testcase>
```

Fig. 3: Example test case of a test suite

```
#include <stdio.h>
#include <unistd.h>
extern char __VERIFIER_nondet_char();

int main() {
  char x = __VERIFIER_nondet_char();
  if (x == 'a') {
    while (1)
      fork();
  } else {
    remove("important.txt");
    if (access("important.txt", F_OK) != -1) {
      return 1;
    }
  }
}
```

Fig. 4: An example program with side effects

A. Test-Suite Exchange Format

TestCov reads and writes test suites in the XML-based exchange format for test suites, which consists of two parts: a metadata file and a set of test-case files, each defining a single test case. The metadata file is an XML file that describes the test suite and is always named `metadata.xml`. Figure 2 shows an example metadata file with all available fields. Some noteworthy fields are: the programming language of the program under test (`<sourcecodelang>`), the coverage criterion the test suite was created for (`<specification>`), the SHA-256 hash of the program under test (`<programhash>`), the program function that is tested by the test suite (`<entryfunction>`), and the system architecture the program tests were created for (`<architecture>`). If the test suite is the result of another test suite, e.g., because of test-suite reduction, the file name of this *input test suite* (`<inputtestsuitefile>`) and its SHA-256 hash (`<inputtestsuitehash>`) can also be recorded. A test-case file (Fig. 3) contains a sequence of tags `<input>` that describe the sequence of input values. The directory structure of test suites is arbitrary, and they are given to and created by TestCov as zip files for efficient storage and convenient handling. Since the exchange format is used in Test-Comp, many test-case generators support the format: CoVeriTest [2], CPA-Tiger[7], Esbmc [10], FairFuzz [13], Klee [7], PRTest [3], Symbiotic [8], and VeriFuzz [9].

B. Test Execution

For test execution, the program under test is compiled against a test harness that consists of two parts: (1) a method `get_input` for receiving test values, and (2) a new definition for each input method in the program under test that delegates to `get_input`.

Method `get_input` reads test inputs from the standard input as C-format strings and parses them into a C type. It supports hexadecimal (e.g., `0x4a`), integer (e.g., `74`), floating point (e.g., `74.5`) and character representation (e.g., `'J'`) for all primitive types (up to long double), as well as single-line string inputs. Methods from the C standard library are used for parsing.

For each input method, a new definition is introduced that calls `get_input` with the corresponding format type of the return type of the input method. For example:

```
int inputMethod() {
  int inputVar;
  get_input("%d", &inputVar);
  return inputVar;
}
```

Given a test suite, TestCov first parses the metadata file to check consistency with the input file and coverage criterion. If one of them is not consistent, the user is informed. Then, TestCov iterates over all test-case files, reads the test inputs for each test, executes the compiled program and sequentially passes the test inputs to the execution via standard input.

To ensure that test execution does not get stuck because of a non-terminating test, a time limit is applied for each test execution. In addition, if a test case contains less input values than necessary, TestCov will terminate the execution once all input values are consumed and a new one is requested.

To ensure that test executions do not alter the system of the user, perform malicious actions, or influence each other, TestCov isolates each test execution in a separate container and control group using runexec[8], a tool provided as part of

[7] https://www.es.tu-darmstadt.de/es/team/sebastian-ruland/testcomp19/

[8] https://github.com/sosy-lab/benchexec/blob/2.0/doc/runexec.md

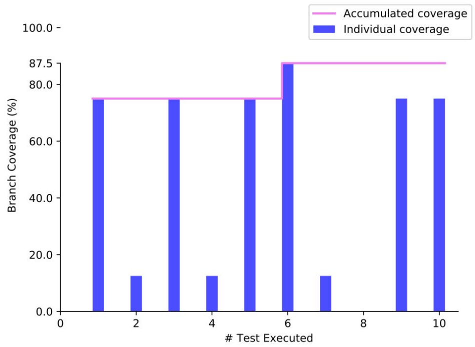


Fig. 5: Plot of individual and accumulated test coverage

BenchExec [4]. We configure it such that test executions in the container have no network access, can not see or modify other system processes, and work on an overlay file system that prevents file modifications in the original system. Files written inside the container are kept in memory and not written to disk. Keeping all file modifications in memory also speeds up test execution in the presence of file operations. Cgroups are a Linux kernel feature that allows to restrict and measure the resource consumption of a process and all its child processes. TestCov uses this to restrict memory usage to a user-specified maximum, to restrict computations to a specified number of CPU cores, and to enforce the time limit on test executions.

Figure 4 shows a program with side effects. The program takes a single character as input, here via Test-Comp-specific method `__VERIFIER_nondet_char`. If the input is ’a’, a *fork bomb* is started that spawns an unbounded number of processes that will eventually fill the process table of the user’s system and make it unusable. Otherwise, the program will delete some file, and check whether the deletion was successful. If TestCov is given a test suite that defines test cases for both branches, it executes both branches properly, but the number of processes is limited (by default to 5000 processes), and the file deletion only happens in the execution’s container, not on the original file system, and thus, no harm is done to the user’s system, while the coverage measurement is still accurate.

C. Coverage Statistics

TestCov provides coverage information per test and for the whole test suite, and creates plots for the coverage criterion. TestCov reads coverage criteria in the query language FQL [11]. Currently, it supports block, branch, and condition coverage, as well as covering calls to an error-function. To compute coverage, TestCov uses GCC instrumentation and LCOV. LCOV stores coverage for each line and program condition in a *tracefile*. LCOV claims to store branch coverage, but considers each condition of a short-circuit boolean operation as a separate branch. For example, the code in Fig. 6 consists of two branches: the if-branch is entered if condition `x > 0 || x < 0` is true, and the (implicit) else branch is entered otherwise. LCOV considers each evaluation of the two conditions `x > 0` and `x < 0` as a separate branch and thus reports that the program has four branches. The evaluation of the first condition (`x > 0`) is always true for that program, so every program execution takes the if-branch. Since condition `x < 0` is never evaluated, LCOV reports a branch coverage of only 25 % instead of the expected 50 %. To circumvent this and implement a proper branch-coverage measurement, TestCov adds program labels `BRANCH_i` at the beginning of each program branch of a program (Fig. 7) and uses the line-coverage measurement of LCOV to check which of the added program labels are covered. This way, TestCov can accurately measure branch coverage.

```
int x = 1;
if (x > 0 || x < 0) {
    // ...
}
```

Fig. 6: Code with short-circuit condition `||`

```
int x = 1;
if (x > 0 || x < 0) {
    BRANCH_1:;
    // ...
} else {
    BRANCH_2:;
}
```

Fig. 7: Code instrumented to compute branch coverage

By default, LCOV stores only the accumulated coverage of all program executions in a single tracefile. To get both the accumulated coverage and the individual coverage of each separate test case, TestCov manages two separate tracefiles with LCOV: a default one that is newly created for each test execution and only stores the coverage of that execution, and one that contains the accumulated coverage over all test executions. While coverage information per test is usually not interesting for mere test execution, it can provide useful insights for test-suite optimization and reduction. TestCov provides coverage statistics as plots and as CSV files that can be easily processed further. It provides a plot (Fig. 5) that shows: (a) the accumulated test coverage (y-axis) after execution of the n-th test (x-axis) of the test suite (step plot, continuous line in Fig. 5), and (b) the test coverage of each test case (bars in Fig. 5). The order of tests in the plot is always the same as the order of execution. It is visible that, for the example, the five tests that reach 75.0 % coverage subsume the three tests that reach 12.5 % coverage, because the accumulated coverage does not increase beyond 75.0 % and 87.5 %, resp., after any of their executions. In addition, it is visible that only the 6th test executed is necessary to achieve the same branch coverage as achieved by all 9 tests of the test suite together, because it provides, on its own, the same coverage as the accumulated coverage of the full test suite.

D. Test-Suite Reduction

TestCov provides test-suite reduction through the strategy design pattern, so different algorithms can be added in the existing infrastructure to reduce a given test suite. By default, TestCov provides the following test-suite reduction technique: If the coverage criterion is to cover calls to an error-function, TestCov creates a new test suite that consists of one test case from the original test suite that covers that error function. If the coverage criterion is to cover lines, branches, or conditions, TestCov creates a new test suite that is potentially smaller than the original test suite and that achieves the same coverage.

To do so, it reads the recorded accumulated coverage after each test execution, and a test is only added to the reduced test suite if its corresponding test execution increased the accumulated coverage. TestCov executes tests in arbitrary order, so this approach does not necessarily produce a minimal test suite, but no additional test executions or computations are necessary for this simple but effective reduction technique.

III. Usage

Installation. TestCov requires Python 3.6 or newer. The following command line installs TestCov and its dependencies (executed from the base directory of the TestCov source code):

```
> python3 setup.py install
```

Execution. TestCov is started via command line, with three required arguments: (1) **–test-suite** to specify the test suite to execute, (2) **–goal** to specify the coverage criterion, and (3) the program file. A test suite is provided as zip file, and a coverage criterion is provided as text file in FQL syntax. The following example command line runs TestCov on test suite `suite.zip`, coverage criterion `criterion.prp`, and program `prog.c`:

```
> testcov –test-suite suite.zip –goal criterion.prp prog.c
```

Directory `output` will contain all output files, i.e., the executable test harness, the reduced test suite, coverage statistics, and plots (in SVG format).

Creation of a separate container for each test execution and coverage measurement increases execution overhead because of the additional file system operations. TestCov provides optional arguments to turn these features off if they are not required. The following command-line prints all such arguments:

```
> testcov –help
```

Adaption of test format. To make adaption of the XML-based test format easy for test-case generators, we provide a small Python library called `tsbuilder`[9]. It can be used to programmatically create test-suite metadata and test cases in the established exchange format for test suites.

IV. Applications

TestCov has been used for Test-Comp'19, where it ran almost 9 million tests created by 9 different test-case generators on 1 720 different programs and 2 different coverage criteria. TestCov was used for both execution and coverage measurement during the competition. All results of the competition are available online.[10] The tables that show results for several test-case generators or meta categories (e.g., `Cover-Branches`) only list the coverage computed by TestCov. The tables for single test generators and sub-categories (e.g., `coverage-branches.ReachSafety-Arrays`-VeriFuzz[11]) provide the full data, including a stripped-down version of plots for accumulated test coverage.

V. Conclusion

TestCov is a tool for test-suite execution on C programs that reads test suites in the simple and standard exchange format of Test-Comp, and performs a robust and reliable test execution. TestCov uses BenchExec, which in turn uses as foundation the containers for isolated execution and control groups for resource control that the operating-system kernel provides. The current version provides both individual and accumulated coverage statistics for four important coverage criteria. TestCov has been successfully used for the execution of Test-Comp'19. While TestCov is implemented for C programs, the used concepts can be easily transferred to other languages.

Acknowledgements

We thank Maximilian Wiesholler for his valuable contributions to the implementation of TestCov.

References

[1] D. Beyer, "Competition on software testing (Test-Comp)," in *Proc. TACAS (3)*, ser. LNCS 11429. Springer, 2019, pp. 167–175. Available: https://www.doi.org/10.1007/978-3-030-17502-3_11

[2] D. Beyer and M.-C. Jakobs, "CoVeriTest: Cooperative verifier-based testing," in *Proc. FASE*, ser. LNCS 11424. Springer, 2019, pp. 389–408. Available: https://doi.org/10.1007/978-3-030-16722-6_23

[3] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Proc. HVC*, ser. LNCS 10629. Springer, 2017, pp. 99–114. Available: https://www.doi.org/10.1007/978-3-319-70389-3_7

[4] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *Int. J. Softw. Tools Technol. Transfer*, vol. 21, no. 1, pp. 1–29, 2019. Available: https://www.doi.org/10.1007/s10009-017-0469-y

[5] D. Beyer and T. Lemberger, "Replication package for article 'TestCov: Robust test-suite execution and coverage measurement' in Proc. ASE '19," Zenodo, 2019. Available: https://doi.org/10.5281/zenodo.3418726

[6] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. ASE*. IEEE, 2008, pp. 443–446. Available: https://doi.org/10.1109/ASE.2008.69

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*. USENIX Association, 2008, pp. 209–224.

[8] M. Chalupa, J. Strejcek, and M. Vitovská, "Joint forces for memory safety checking," in *Proc. SPIN*. Springer, 2018, pp. 115–132. Available: https://www.doi.org/10.1007/978-3-319-94111-0_7

[9] A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, "VeriFuzz: Program-aware fuzzing (competition contribution)," in *Proc. TACAS (3)*, ser. LNCS 11429. Springer, 2019, pp. 244–249. Available: https://doi.org/10.1007/978-3-030-17502-3_22

[10] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of C programs via k-induction," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, no. 1, pp. 97–114, Feb. 2017. Available: https://www.doi.org/10.1007/s10009-015-0407-9

[11] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "Query-driven program testing," in *Proc. VMCAI*, ser. LNCS 5403. Springer, 2009, pp. 151–166. Available: https://doi.org/10.1007/978-3-540-93900-9_15

[12] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015. Available: https://doi.org/10.1109/TSE.2015.2454513

[13] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. ASE*. ACM, 2018, pp. 475–485. Available: https://www.doi.org/10.1145/3238147.3238176

[9] https://gitlab.com/sosy-lab/software/test-format/tree/v2.0/python_modules/tsbuilder

[10] https://test-comp.sosy-lab.org/2019/results/

[11] https://test-comp.sosy-lab.org/2019/results/results-verified/verifuzz.2019-02-06_0717.results.test-comp19_prop-coverage-branches.ReachSafety-Arrays.xml.bz2.merged.xml.bz2.table.html

Tests from Witnesses

Execution-Based Validation of Verification Results

Dirk Beyer[1], Matthias Dangl[1], Thomas Lemberger[1],
and Michael Tautschnig[2]

[1] LMU Munich, Munich, Germany
[2] Queen Mary University of London, London, UK

Abstract. The research community made enormous progress in the past years in developing algorithms for verifying software, as shown by international competitions. Unfortunately, the transfer into industrial practice is slow. A reason for this might be that the verification tools do not connect well to the developer work-flow. This paper presents a solution to this problem: We use verification witnesses as interface between verification tools and the testing process that every developer is familiar with. Many modern verification tools report, in case a bug is found, an error path as exchangeable verification witness. Our approach is to synthesize a test from each witness, such that the developer can inspect the verification result using familiar technology, such as debuggers, profilers, and visualization tools. Moreover, this approach identifies the witnesses as an interface between formal verification and testing: Developers can use arbitrary (witness-producing) verification tools, and arbitrary converters from witnesses to tests; we implemented two such converters. We performed a large experimental study to confirm that our proposed solution works well in practice: Out of 18 966 verification results obtained from 21 verifiers, 14 727 results were confirmed by witness-based result validation, and 10 080 of these results were confirmed alone by extracting and executing tests, meaning that the desired specification violation was effectively observed. We thus show that our approach is directly and immediately applicable to verification results produced by software verifiers that adhere to the international standard for verification witnesses.

1 Introduction

Automatic software verification, i.e., using methods from program analysis and model checking to find out whether a program satisfies or violates a given specification, is a successful and mature technology. The efficiency and effectiveness of the available verification tools for C programs is shown in the annual competition on software verification [5]. Despite this success story in research, the state-of-the-art in practice is that not many software projects have such verification tools incorporated into their software-development process. The reason for this gap between availability of technology on the one side and missed opportunities on the other side is perhaps twofold: (a) developers are frustrated by false alarms, i.e.,

© Springer International Publishing AG, part of Springer Nature 2018
C. Dubois and B. Wolff (Eds.): TAP 2018, LNCS 10889, pp. 3–23, 2018.
https://doi.org/10.1007/978-3-319-92994-1_1

in the past, static analyzers reported too many bugs that were not observable in a concrete program execution, and thus, developers have lost confidence in bug reports [20]; (b) there is a lack of appropriate interfacing, i.e., it is difficult for developers to leverage advantages of the verification tools because they are difficult to integrate and difficult to learn from [1].

To overcome these two problems, we propose (i) to use verifiers that produce verification witnesses, i.e., abstract descriptions of one or more paths to a specification violation (many such tools are already available [1]), and (ii) to validate whether a real bug has been found by constructing a test from the produced verification witness and observing the execution of that test. This way, issue (a) above is solved because, if the test execution does show and thus confirm the reported specification violation, the verification result can be examined with high confidence and on a concrete, executable example (e.g., with a debugger), and issue (b) is solved because we bridge the gap between the, in most projects, unfamiliar domain of verification and the established domain of testing, which makes it easier to integrate verification into the development process.

Execution-Based Validation of Witnesses. Witness validation based on model-checking technology works well [4,5,9,14], but the disadvantage is that due to over-approximation, the validation might be as imprecise as the verification step. A verification witness serves as a (potentially coarse) description of a part of the state space of a program that contains a specification violation, and the witness validators can confirm or reject the error report. We complement the witness-validation technology by direct test execution: A test case (e.g., unit-test code) is built from the violation witness, and this test case provides a precise and transparent way to confirm and examine it. [2] By observing and analyzing an execution that exposes undesirable behavior, developers can convince themselves that the error report is correct, and address the reported bugs without the risk of wasting time on a false alarm. If the execution does not violate the specification, the witness might have represented a false alarm and the developer can assign a lower priority to that report.

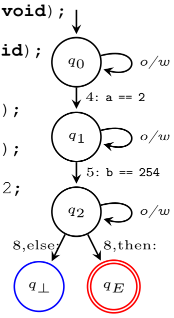
Witnesses as Communication Interface. One barrier for the adoption of verification technology is that developers have to spend considerable time on understanding a verification tool and on becoming familiar with it. Thus, we have to avoid the "lock-in" effect: people might not want to decide for one particular tool if they have to invest time again when they wish to change the decision later. If the developer constructs the integration on top of the exchangeable verification witnesses, i.e., using the witnesses as interface to the verification tools, the verification tool is exchangeable without any change to the testing process. [3]

[1] https://sv-comp.sosy-lab.org/2017/systems.php

[2] It has been shown that model checkers can be effective in constructing useful tests [12].

[3] At least 21 verifiers are available that produce witnesses in the exchangeable format (cf. Table 1, which lists the verifiers that we use in our experiments).

```c
extern void __VERIFIER_error(void);
extern unsigned char __VERIFIER_nondet_uchar(void);
int main(void) {
  unsigned char a = __VERIFIER_nondet_uchar();
  unsigned char b = __VERIFIER_nondet_uchar();
  unsigned char sum = a + b;
  unsigned char mean = sum / 2;
  if (mean < a / 2) {
    __VERIFIER_error();
  }
  return 0;
}
```



```c
#include <stdlib.h>
void __VERIFIER_error() { exit(107); }
unsigned char __VERIFIER_nondet_uchar() {
  static unsigned int test_vector_index = 0;
  unsigned char retval;
  switch (test_vector_index) {
    case 0: retval = 2U; break;
    case 1: retval = 254U; break;
  }
  ++test_vector_index;
  return retval;
}
```

(a) Example program **(b)** Witness automaton **(c)** Injection of test values

Fig. 1. An incorrect example C program (a), the corresponding violation witness produced by the verifier (b), and a code fragment used to inject the extracted test values for compilation (c)

Tests from Witnesses. In order to flexibly bridge the gap from witness to test, we provide two independently developed implementations of tools that take as input a program and a violation witness, and synthesize a test that is compilable and executable. This approach provides the following three features: (1) the result of a verification tool can be validated by compiling and executing the corresponding test—if the test violates the specification, the verification tool reported a correct alarm and the result can be handled appropriately; (2) the synthesized unit tests can be stored and maintained together with the other unit tests, but can also be re-constructed at any time on demand; (3) independently from the verification tool that produced the witness, the full repertoire for inspecting a failing program—such as debuggers, profilers, and visualization tools—can be used by the developer to understand the bug that the test represents.

Experimental Study. To evaluate our proposal, we performed experiments on thousands of witnesses. We took many C programs from the largest public repository of verification tasks and many witness-producing verification tools, and collected 13 200 witnesses of specification violations. We obtained another 5 766 refined witnesses using witness refinement, a procedure introduced in the original work on verification witnesses [9]. This technique is supposed to refine witnesses to be more concrete, so we should be able to generate better test cases from them. In conjunction with the two existing validators, CPAchecker and Ultimate Automizer, our method significantly increases the confirmation rate: out of the total of 18 966 witnesses, we were able to extract test cases for 10 080 of them, meaning that we successfully created and executed the tests, and the specification violation was observed. Using the new approach, we increased the confirmed results from 12 821 to 14 727 in total.

Example. In the following, we illustrate the complete process from running a verification task using a verifier through synthesizing the test code from the violation witness to compiling the program and executing it.

Figure 1a shows a program that attempts to calculate the mean of two integer numbers, a computation that is often required in binary-search algorithms. In lines 4 and 5, two variables `a` and `b` of type `unsigned char`[4] are initialized nondeterministically, for example from user input. The subsequent lines are supposed to calculate the mean of the two variables, by first computing their sum in line 6 and then dividing it by `2` in line 7. If the mean of `a` and `b` has been calculated correctly, it must not be less than half of either of the two values. This condition is asserted in lines 8 to 10. We can check whether the condition is satisfied by specifying that the function `__VERIFIER_error()` must not be reachable, and then running a verifier on this verification task. The verifier should detect and report that the assertion will be violated if the sum of `a` and `b` exceeds the range of the data type `unsigned char`, causing an overflow. Figure 1b shows a violation-witness automaton [9] that represents a counterexample to the specification. The automaton specifies that if we assume that `a` is assigned the value `2` in line 4 and `b` is assigned the value `254` in line 5, control will flow to the `then`-branch in line 8, causing a violation of the specification. To independently validate this witness, we can then extract the input values for `a` and `b`, and use them to provide an implementation of the input function `__VERIFIER_nondet_uchar()` and the `__VERIFIER_error()` function as depicted in Fig. 1c. After compiling Fig. 1a and 1c into an executable and running it, we can confirm that these input values trigger the call to `__VERIFIER_error()` by checking its return code. We can even use a debugger such as GDB to step through the compiled program and observe the faulty behavior directly. The debugger will show that the sum of `a` and `b`, respectively `2` and `254`, computed in line 6 wraps around to `0`. Therefore, the mean is incorrectly calculated as `0` in line 7. The condition in line 8 then evaluates to `1`, because `0` is smaller than `1`.

It must be noted that the witness depicted in Fig. 1b is very precise: it provides a concrete counterexample with explicit values for `a` and `b`. But in general, a violation witness may simply describe a part of the state space that contains a specification violation, i.e., an abstract counterexample. Suppose a verifier is only able to provide a witness that specifies that if `a + b` is greater than `255` in line 6, the specification will be violated. By using witness refinement [9], we can obtain from this abstract witness a concrete witness like Fig. 1b.

Contributions. Our approach features the following advantages:

- Verification tools sometimes produce false alarms, which can lead to severe waste of investigation time. We synthesize tests from verification witnesses, and consequently trust only verification results confirmed by test execution.
- There are several witness-based validators available, but our execution-based validation of the error path can be more precise and more efficient, compared to the previously available validators.
- Avoidance of technology lock-in: A developer's work flow does not depend on a particular choice of verification tool, because the developer's infrastructure hooks in at the witness. The developer may elect to use a different verifier, or even use multiple verifiers simultaneously—at no additional cost.

[4] The example also works for larger data types, but for ease of presentation, we aim to keep the range of values small, so that all calculations can be followed by hand.

- Compared to working with witnesses, developers are more familiar with tests, and more supporting tools—such as profilers, memory analyzers, and visualization tools—are available to analyze the tests that correspond to the witnesses.
- The newly generated tests can complement the existing test suite, and the tests as well as the witnesses can be stored and maintained as first-class objects in the software life cycle.

Related Work. Our approach is based on a number of existing ideas, which we outline in the following.

Verification Witnesses. We build our contributions on top of existing work on violation witnesses [9], which we will describe in more detail in the background section. The problem that verification results are not treated well enough by the developers of verification tools is known and there are also other works that address the same problem, for example, the work on execution reports [18].

Test-Case Generation. The idea to generate test cases from verification counterexamples is more than ten years old [6,48], has since been used to create debuggable executables [39,42], and was extended and combined to various successful automatic test-case generation approaches [25,27,36,46]. We complement existing techniques in the following ways: Our technique works on the flexible exchange format for violation witnesses. In case such a witness constitutes only an abstract counterexample, we can use witness refinement to efficiently obtain a concrete one [9]. Such a mechanism is not available for existing test-case generation tools.

Execution. Other approaches [16,22,35] focus on creating tests from concrete and tool-specific counterexamples. In contrast, our approach does not require full counterexamples, but works on more flexible, possibly abstract, violation witnesses.

Debugging and Visualization. Besides executing a test, it is important to understand the cause of the error path, and there are tools and methods to debug and visualize program paths [3,7,28].

2 Background

A verification witness is an exchangeable object that stores valuable information about the verification process and the verification result. The key is that the format is open and exchangeable, and that many verification tools support it.

Witness Construction. It has been commonly established practice for verifiers to provide a counterexample to witness a specification violation, in particular since counterexamples were used to refine abstract models [21]. The problem was that these counterexamples were more or less 'dumps' of paths through the state space, sometimes not human-readable, sometimes not machine-readable. Recent efforts of the software-verification community established a common exchange format for verification results as verification witnesses [9]. In this format, a so-called violation-witness automaton (as seen in Fig. 1b) describes a state space that contains the specification violation. This state space does not necessarily have to

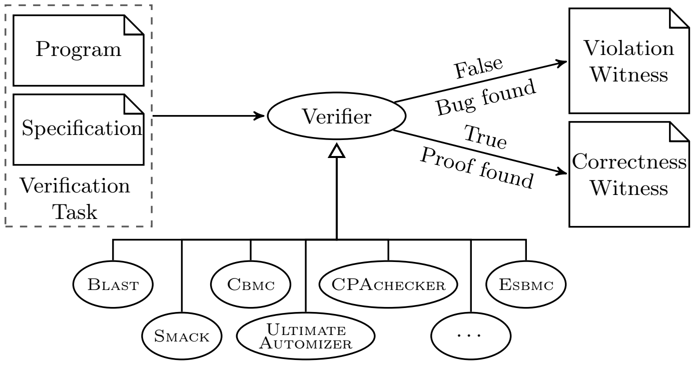
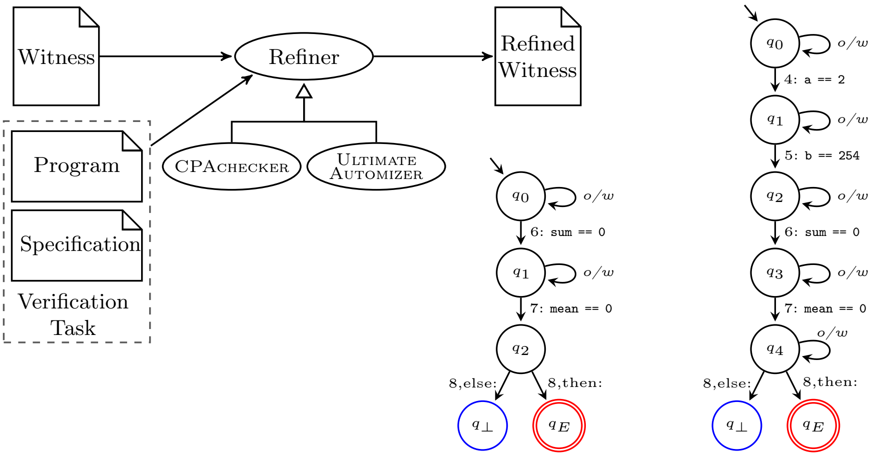


Fig. 2. Software verifiers produce witnesses



(a) Concept sketch **(b)** Abstract witness **(c)** Refined witness

Fig. 3. Concept of witness refinement with example abstract and refined witnesses for the example program depicted in Fig. 1a from the introduction

represent just a single error path, but may contain multiple error paths and even paths without a specification violation. As an example for the use of verification witnesses, the International Competition on Software Verification (SV-COMP) applies this format and counts a report of a found bug only if a corresponding violation witness is reported and confirmed [4]. Figure 2 illustrates the process: the verifiers can be exchanged according to the needs of the user, there is no risk of technology lock-in. Figure 2 also shows that the exchange format for witnesses has recently been extended to correctness witnesses [8]. In the remainder of this paper, however, we will only consider violation witnesses.

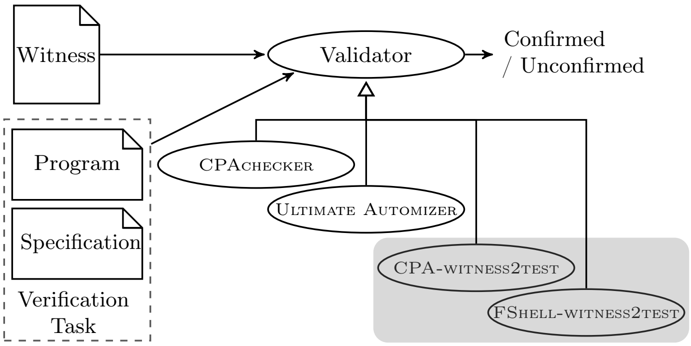
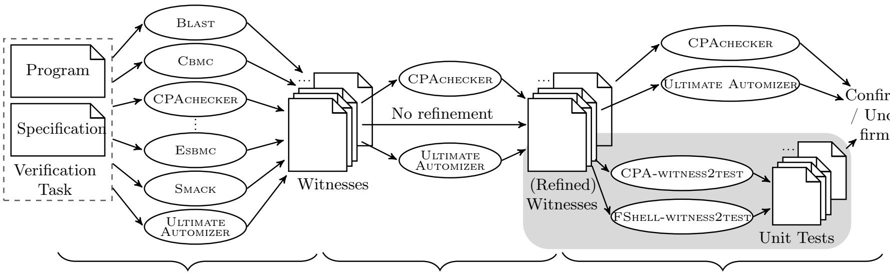


Fig. 4. Violation-witness validation

Witness Refinement. The original work on verification witnesses [9] contains the proposal to consider refinement of witnesses. The idea is to take a violation witness as input, replay it with a validating verifier, and produce a new witness that is more detailed. A more detailed violation witness is closer to a concrete program path and makes the validation process faster. We will later in this paper use an instance of a witness refiner to improve witnesses from other verification tools towards being able to successfully derive tests from witnesses. Figure 3a illustrates the optional step of using witness-refining validators to strengthen a witness. Figure 3b shows another, valid violation witness for the previously considered program from Fig. 1a. In contrast to the witness in Fig. 1b, this witness does not specify any concrete values for the two nondeterministic values of variables `a` and `b`, but specifies that a property violation occurs if the intermediate variables `sum` and `mean` are both equal to `0`. This witness automaton represents a set of 256 different counterexamples: every counterexample with values for `a` and `b`, so that `a + b == 0` during execution. Figure 3c shows a violation witness that is a refinement of the more abstract witness in Fig. 3b that additionally specifies concrete values for the two variables `a` and `b` and thus restricts the search space in witness validation early on.

Witness Validation. Violation witnesses can be used to independently re-establish the verification result by using a witness-based result validator that takes the information from the witness to find a path through the state space of the program to a specification violation. Thus, a successful validation increases trust in the verification result, and developers no longer need to rely on the verifiers alone. Instead, they can focus their attention on the validated results and assign a lower priority to unconfirmed alarms. The existing witness-based result validators employ potentially-expensive model-checking techniques to replay error paths that are represented in the witness. While this is a powerful technique (it can reconstruct error paths even for abstract witnesses), the technique still has the limitations of common program-analysis and model-checking techniques, namely that the technique may over-approximate the semantics of the programming language,



(a) Witness construction **(b)** Optional witness refinement **(c)** Witness validation

Fig. 5. Software verification with witnesses: construction, (optional) refinement, and validation work flow

thus potentially confirming false alarms or rejecting valid violation witnesses. As a solution to this, we propose an execution-based approach to witness-based result validation. Figure 4 shows the two existing validators CPAchecker and Ultimate Automizer together with the two new, execution-based validators that we introduce in this paper: CPA-witness2test and FShell-witness2test.

3 Tests from Witnesses

This section introduces a new, yet unexplored, application of witnesses that can easily be integrated into established processes for verification-result validation, as summarized by Fig. 5. The highlighted area in Fig. 5 outlines the goal: for a given violation witness, we want to construct a test that can be compiled and executed to check that the bug is realizable. In particular, driven by our desire to keep the work-flow independent from special verifiers, we want to have two independently developed implementations of such witness-to-test tools.

Our new, execution-based witness validator does not require the aid of model-checking techniques for validating verification results: we generate a test harness (test code for the program), which can be compiled and linked together with the original subject program and executed. If the execution does not trigger the described bug, the witness is deemed spurious, i.e., not realizable.

Adding this new tool to the pool of available witness-based result validators not only increases the diversity of validation techniques and its potential for establishing trust in verification results, but also adds novel features to the validation process: As a valuable by-product of a successful validation, the developers are able to obtain executable test code that is guaranteed to reproduce the bug in their system, and they can use all of the infrastructure for inspecting and debugging that they are trained and experienced in and that is already in place in their development environment. For example, a C developer might simply run GDB to step through the executable error path.

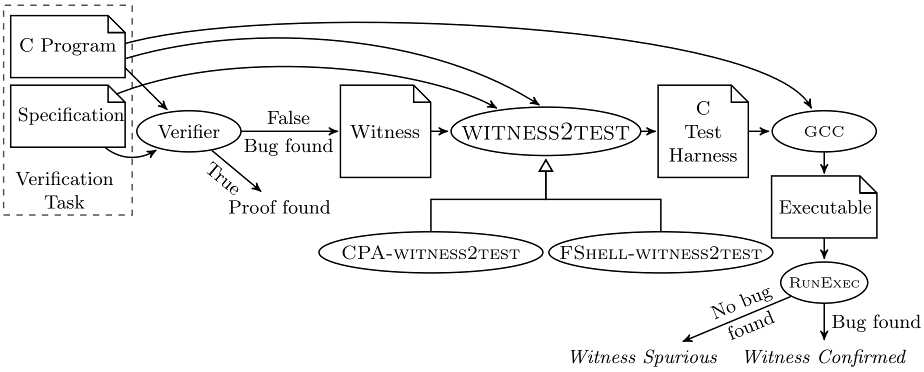


Fig. 6. Flow of execution-based result validation

Figure 6 shows the complete picture of execution-based witness validation. The verification task (a given program with a given specification) is verified by a chosen verifier. If the verifier reports a specification violation (FALSE, bug found) it also produces a violation witness. (Our work does not consider the outcome TRUE, for which the development of practical support, such as correctness witnesses [8] and compact proof witnesses [32], is also a subject of ongoing research.) The witness in GraphML format [15] is then given to WITNESS2TEST, which synthesizes a test harness that drives the program to the specification violation. In order to support our claim of independence from any particular tool implementation, we implement two completely different instances of WITNESS2TEST, namely CPA-WITNESS2TEST (based on open-source components from CPACHECKER) and FSHELL-WITNESS2TEST (based on ideas from FSHELL). The test-harness and the original (unchanged) program are then compiled and linked to obtain an executable program. The executable program is then executed in a safe execution container. [5] If the reported specification violation is observed during this execution, the witness is confirmed. Otherwise the witness is not confirmed, most likely because the witness is not precise enough or even spurious.

3.1 CPA-WITNESS2TEST

One of our implementations for the WITNESS2TEST component of the architecture outlined in Fig. 6 is CPA-WITNESS2TEST, which is based on the CPACHECKER framework [11]. For our purpose of matching an input witness to the program source code of a verification task and generating a *test harness*, we configure CPACHECKER to use the witness automaton as a *protocol automaton* [9] to guide and restrict the state-space exploration to the program paths that the witness represents. Unlike observer automata [44], which we use to represent the specification and which can only monitor the state-space exploration of an analysis, *protocol automata* may also restrict the state-space exploration, for example to a specific program path,

[5] We choose BENCHEXEC [13] as container solution, because it is also used by SV-COMP.

thereby guiding the analysis along that path. In our case, this path is the error path represented by the protocol automaton. We configure the analysis to only consider the (syntactical) branching information of the *protocol automaton* and to not semantically analyze the path. During this *protocol analysis*, we observe which input-value assumptions from the witness correspond to which input function or variable of the program. By collecting this information, we are able to construct a *test vector* for the program. The *test vector* maps an input value to each input variable and a list of input values to each external function. We synthesize a *test harness* from a test vector by providing initializations for input variables and definitions for external functions. An external function with a list $(v_0, \ldots, v_{n-1})$ of $n \in \mathbb{N}$ input values is defined by using a `switch` statement with n cases over a static counter variable $0 \leq i < n$ that is initialized to 0 and incremented after each call to the function. Each case of the `switch` statement corresponds to an input value, such that `case` i selects v_i. We also inject a call to the `exit` function so that when we later execute the program, we can detect that the intended violation of the specification was triggered, i.e., the program crashed precisely due to the bug described by the witness, by checking for a specific execution return value. Figure 1c shows the `exit(107)`-call in line 2 and a definition of an input function `__VERIFIER_nondet_uchar()` in lines 3 to 12 as generated by CPA-witness2test, where the counter variable `test_vector_index` represents i. The `switch` statement in this function definition provides sequential access to the two input values $(2, 254)$ that CPA-witness2test extracted from the witness of Fig. 1b for the program shown in Fig. 1a.

3.2 FShell-witness2test

The key design principle of FShell-witness2test is independence from existing verification infrastructure: FShell-witness2test's results shall—by design—be unbiased towards any existing software-analysis framework. While this does imply limitations on the class of witnesses that can be processed as discussed below, it does yield further advantages: FShell-witness2test is easy to extend for prototyping, and does not require any background in software verification.

FShell-witness2test comprises two major parts: (1) A Python-based processor of the witness and the input program, using **`pycparser`**[6] to generate test vectors in a format compatible with FShell [31]. (2) A Perl script that translates such test vectors into a test harness.

For a given verification task and witness, FShell-witness2test first parses the specification to restrict itself to reachability properties (call to error function should not be reachable). The witness and the C program are then handed to the Python-based processor. The specification defines the entry function to be used by the generated test harness.

As **`pycparser`** cannot handle various GCC extensions, input programs are preprocessed and sanitized by performing text replacement and removal. We then obtain the abstract syntax tree and iterate over its nodes to gather data types and

[6] https://github.com/eliben/pycparser

source locations of (1) all procedure-local uninitialized variables, (2) all functions with prefix __VERIFIER_nondet, and (3) all uses of such functions. We refer to the locations of uninitialized variables and nondeterministic-input function uses as *watch points*.

Finally we build a linear sequence of nodes from the GraphML encoding of the witness. Traversing this sequence, any match of line numbers against the watch points triggers an attempt to extract values from assumptions in the witness. If parsing the C code that is contained in the assumption succeeds, then an input value is recorded.

The test vector is compatible with the output of FShell; the program of Fig. 1 yields the following test vector:

```
IN:
  ENTRY main()@[file mean.c line 1]
  unsigned char __VERIFIER_nondet_uchar()@[file mean.c line 4]=2
  unsigned char __VERIFIER_nondet_uchar()@[file mean.c line 5]=254
```

Such a test vector is translated to a Makefile that generates an actual test harness, which consists of invocation code and the implementation of various nondeterministic-input functions that are present in the program. FShell-witness2test reports False (confirming the violation) if, and only if, the property violation is detected in the output of the test execution.

4 Evaluation

We perform a large experimental study to demonstrate the general applicability and the advantages of our approach.

4.1 Evaluation Goals

The goal of our experimental evaluation is to collect experience with our new kind of result validation and to support the following claims with data for a large set of witnesses:

Claim 1: Execution-based validators can confirm violation witnesses that the existing validators (which are based on model-checking technology) can not validate. Thus, execution-based validation increases the overall effectiveness.

Claim 2: Result validation based on executable tests can be faster than result validation based on model-checking technology.

Claim 3: Violation witnesses in the common exchange format for verification results (cf. Sect. 2) are a valuable source to synthesize test code for specification violations to complement existing test suites.

4.2 Experiment Setup

We used the benchmarking framework BenchExec (revision `fb32a3e7`) to conduct our experiments. In order to experimentally evaluate our approach, we first construct a large set of witnesses that is diverse in terms of (a) subject programs and (b) verification tools that create witnesses.

Subject Programs. For (a), we consider the largest available set of verification tasks [7] from the community of automatic software verification and select all 5 692 verification tasks with a reachability property [8].

Verifiers. For (b), we use all verification tools that participated in SV-COMP 2017 for property *ReachSafety* and whose license allows us to use it [9]. Table 1 lists all verifiers that we executed to produce violation witnesses. The table lists in the first column the verifier name with a link to the project web site for more information, and a reference to the paper describing the corresponding verifier. For the experiments, we took the archives from the competition web site. [10]

Collection of Witnesses. From the given verification tasks and verifiers, we started verification runs and collected the obtained violation witnesses. For this replication of the SV-COMP experiments we followed thoroughly the description on the competition web site [10] and in the report [4]. In particular, we started each verifier only on those verification tasks and with those parameters that were declared by the development teams of the verifiers [11]. The number of witnesses that we obtained with this process is reported in Table 1 (col. 'Unref.'). Because we use all available verifiers (not only those that performed well in the competition), the set of witnesses contains also bad witnesses (e.g., that are syntactically incorrect). We did not want to exclude them for external validity.

To further increase the external validity of our evaluation, we additionally produced witnesses by applying a witness-refinement technique (cf. Sect. 2) to 13 200 witnesses above. We used the witness-refiner from the CPAchecker framework for this step. This refinement is often able to improve imprecise witnesses by adding concrete input values, and yields another 5 766 witnesses (col. 'Ref.') to a total of 18 966 witnesses (col. 'Total') that we will run our experiments on.

In order to highlight the differences between model-checking-based validation approaches and execution-based validation approaches, we manually crafted some verification tasks and corresponding witnesses. These witnesses allow us a more detailed discussion of some effects, but were not added to our set of automatically generated witnesses.

Computing Resources. Our experiments were conducted on machines with an Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 16.04 with Linux kernel 4.4). We limited the verification runs to four processing units (i.e., two physical cores), 7 GB of memory, and 15 min of CPU time, and the

[7] https://github.com/sosy-lab/sv-benchmarks/tree/423cf8c

[8] We have to restrict the experiments to property *ReachSafety* because there were no witness validators available for the other properties.

[9] There are also two commercial verifiers that produce witnesses, but we cannot use them due to their proprietary license.

[10] https://sv-comp.sosy-lab.org/2017/systems.php

[11] https://github.com/sosy-lab/sv-comp/tree/svcomp17/benchmark-defs

Table 1. Violation witnesses produced by verifiers and resulting tests

Verifier		Produced witnesses			Produced tests			
		Unref.	Ref.	Total	Count	kLOC	kB	# Inputs (Avg.)
2LS	[45]	992	384	1 376	1 208	89.9	3 999	7.57
BLAST	[47]	778	202	980	327	29.0	938	0.271
CBMC	[34]	831	467	1 298	1 249	67.7	2 991	6.33
CEAGLE		619	426	1 045	540	92.2	262	5.39
CPA-BAM-BnB	[2]	851	175	1 026	158	42.9	1 114	0
CPA-kInd	[10]	263	193	456	656	56.2	2 967	14.9
CPA-Seq	[23]	883	767	1 650	838	95.5	3 895	1.79
DepthK	[43]	1 159	305	1 464	1 302	65.4	3 170	2.96
Esbmc	[37]	653	148	801	478	21.0	1 983	2.53
Esbmc-falsi	[37]	981	395	1 376	1 133	53.7	1 906	1.81
Esbmc-incr	[37]	970	392	1 362	1 126	53.5	1 896	1.82
Esbmc-kInd	[24]	847	352	1 199	1 028	48.9	1 774	1.69
Forester	[30]	51	0	51	0	0	0	-
PredatorHP	[33]	86	61	147	80	17.2	434	0
Skink	[17]	30	25	55	44	0.290	8	0
Smack	[41]	871	632	1 503	1 576	128	5 654	6.09
Symbiotic	[19]	927	411	1 338	589	38.1	1 375	0
SymDIVINE	[38]	247	224	471	405	13.4	580	0
UAutomizer	[29]	514	70	584	121	2.24	59	0
UKojak	[40]	309	67	376	116	2.15	55	0
UTaipan	[26]	338	70	408	121	2.23	59	0
Total		13 200	5 766	18 966	13 095	920	35 119	5.60

witness-refinement and validation runs to two processing units (i.e., one physical core), 4 GB of memory, and 1.5 min of CPU time. All CPU times are reported with two significant digits. The limits are inspired by SV-COMP.

Validators. We used CPA-witness2test in version 1.6.14-tap18 from CPA-checker and FShell-witness2test in revision `2a76669f` from the `test-gen` branch. We used the model-checking based witness validators CPAchecker, version 1.6.14-tap18, and Ultimate Automizer 0.1.8.

4.3 Availability of Data and Tools

All tools and all data obtained in our experiments are available via our supplementary web page. [12] The verification tasks are also publicly available [7].

4.4 Results

Claim 1: Effectiveness. Table 2 reports the number of witnesses that the individual validators were able to confirm. In the columns, it shows: the results of

[12] https://www.sosy-lab.org/research/executionbasedwitnessvalidation/

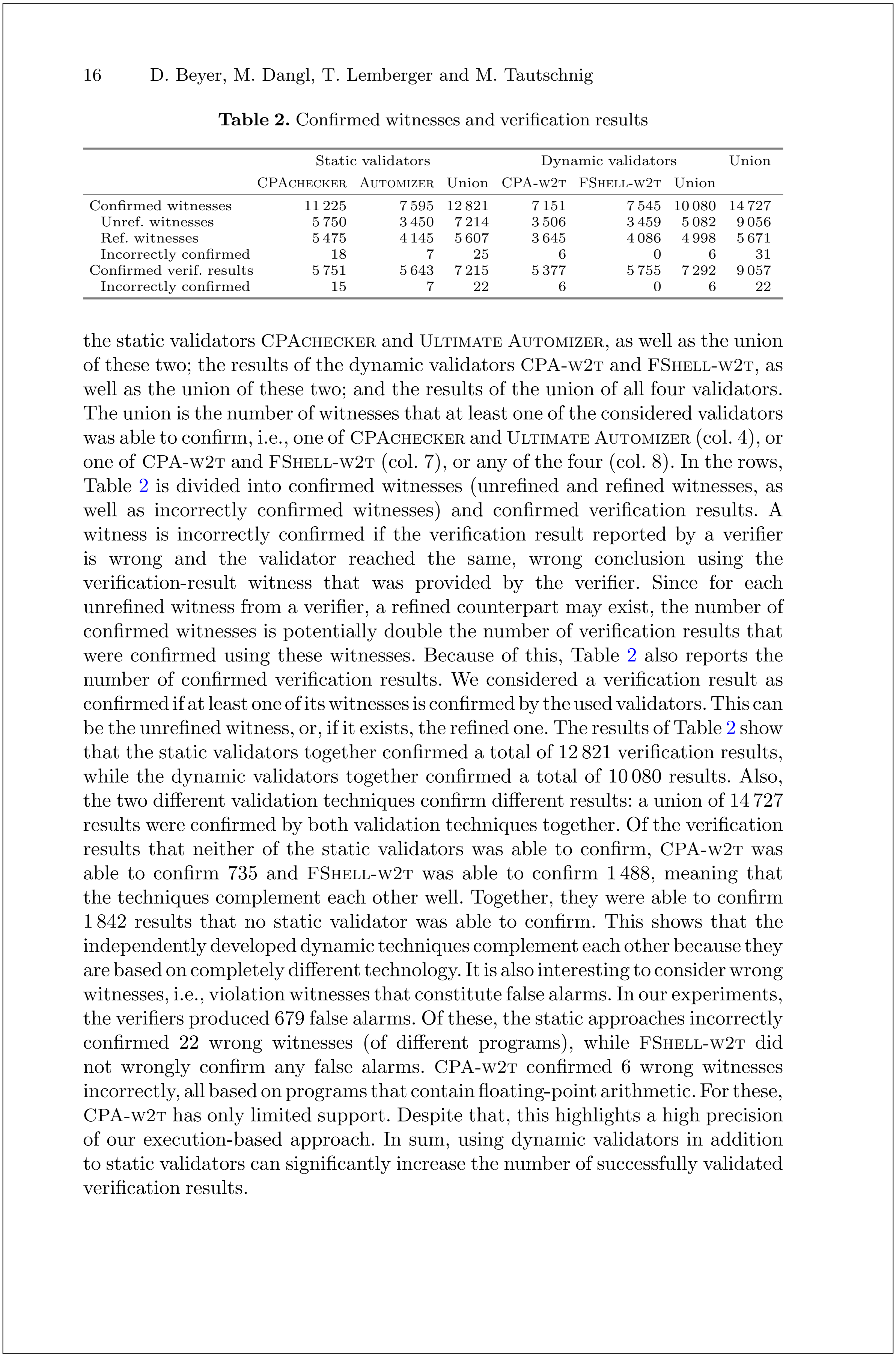


Table 2. Confirmed witnesses and verification results

	Static validators			Dynamic validators			Union
	CPAchecker	Automizer	Union	CPA-w2t	FShell-w2t	Union	
Confirmed witnesses	11 225	7 595	12 821	7 151	7 545	10 080	14 727
Unref. witnesses	5 750	3 450	7 214	3 506	3 459	5 082	9 056
Ref. witnesses	5 475	4 145	5 607	3 645	4 086	4 998	5 671
Incorrectly confirmed	18	7	25	6	0	6	31
Confirmed verif. results	5 751	5 643	7 215	5 377	5 755	7 292	9 057
Incorrectly confirmed	15	7	22	6	0	6	22

the static validators CPAchecker and Ultimate Automizer, as well as the union of these two; the results of the dynamic validators CPA-w2t and FShell-w2t, as well as the union of these two; and the results of the union of all four validators. The union is the number of witnesses that at least one of the considered validators was able to confirm, i.e., one of CPAchecker and Ultimate Automizer (col. 4), or one of CPA-w2t and FShell-w2t (col. 7), or any of the four (col. 8). In the rows, Table 2 is divided into confirmed witnesses (unrefined and refined witnesses, as well as incorrectly confirmed witnesses) and confirmed verification results. A witness is incorrectly confirmed if the verification result reported by a verifier is wrong and the validator reached the same, wrong conclusion using the verification-result witness that was provided by the verifier. Since for each unrefined witness from a verifier, a refined counterpart may exist, the number of confirmed witnesses is potentially double the number of verification results that were confirmed using these witnesses. Because of this, Table 2 also reports the number of confirmed verification results. We considered a verification result as confirmed if at least one of its witnesses is confirmed by the used validators. This can be the unrefined witness, or, if it exists, the refined one. The results of Table 2 show that the static validators together confirmed a total of 12 821 verification results, while the dynamic validators together confirmed a total of 10 080 results. Also, the two different validation techniques confirm different results: a union of 14 727 results were confirmed by both validation techniques together. Of the verification results that neither of the static validators was able to confirm, CPA-w2t was able to confirm 735 and FShell-w2t was able to confirm 1 488, meaning that the techniques complement each other well. Together, they were able to confirm 1 842 results that no static validator was able to confirm. This shows that the independently developed dynamic techniques complement each other because they are based on completely different technology. It is also interesting to consider wrong witnesses, i.e., violation witnesses that constitute false alarms. In our experiments, the verifiers produced 679 false alarms. Of these, the static approaches incorrectly confirmed 22 wrong witnesses (of different programs), while FShell-w2t did not wrongly confirm any false alarms. CPA-w2t confirmed 6 wrong witnesses incorrectly, all based on programs that contain floating-point arithmetic. For these, CPA-w2t has only limited support. Despite that, this highlights a high precision of our execution-based approach. In sum, using dynamic validators in addition to static validators can significantly increase the number of successfully validated verification results.

Table 3. Performance comparison for witnesses that all validators confirmed (CPU time for 2 685 witnesses)

	CPAchecker	Automizer	CPA-w2t	FShell-w2t
Total time (s)	20 000	45 000	30 000	1 900
Average time (s)	7.4	17	11	0.72
Median time (s)	6.2	11	5.9	0.71

Claim 2: Efficiency. Table 3 considers only results that were confirmed by all validators, to compare the execution performance. For the dynamic validators, the reported run time contains all three steps: generating the test from the witness, compiling and linking, and executing the test. The results show that the static approaches are slow (CPAchecker and Ultimate Automizer), that the approach that assembled a static analysis for test generation from CPAchecker components is also slow (CPA-w2t), and that the light-weight implementation that is specifically tailored to generating tests from witnesses is extremely fast (FShell-w2t). Figure 7 displays quantile functions that show for each validator the necessary maximum CPU time (y-axis) for confirming a certain quantile of results (x-axis). We observe that FShell-w2t significantly outperforms all other validators.

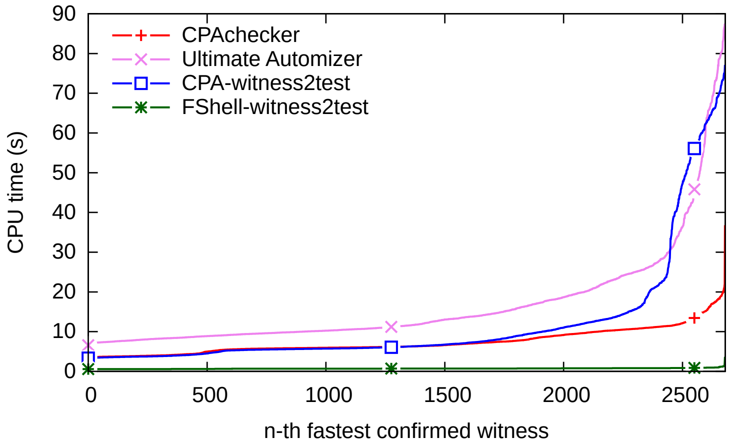


Fig. 7. Quantile plot for CPU time consumed for validating witnesses accepted by all validators

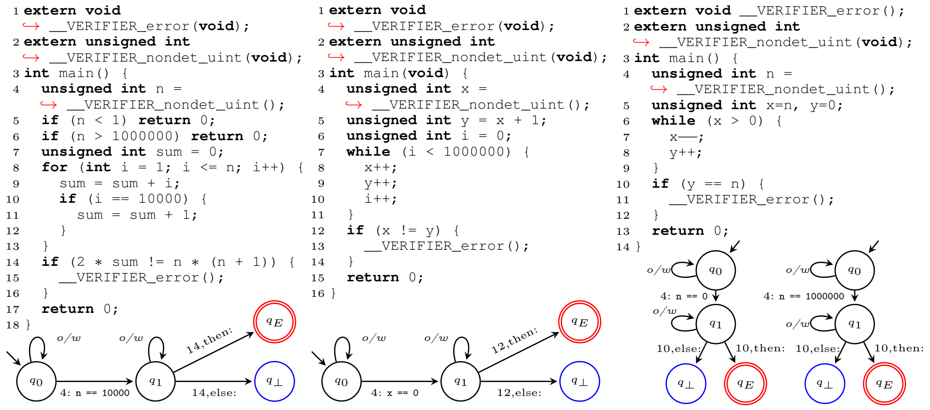
Interestingly, in our validation we observed that the witnesses that require the most time to validate are witnesses that are large in size and that describe a long, detailed error path. Most of these are produced by verifiers that use bounded model checking, e.g., Cbmc and CPA-kInd, or by our refinement step.

Claim 3: Test Generation. The last four columns of Table 1 relate the number of witnesses that we processed to the number of produced tests for which failing executions are realizable. With 'produced tests' we refer to the tests that were produced by any of the dynamic validators and for which the test execution lead to an observed specification violation. Note that because we collect tests from both dynamic validators, the numbers of produced tests exceed the number of witnesses in some rows. Since the tests are available in source code, and could be maintained and re-used by developers in practical application scenarios, we also report the size of these unit tests in lines of code, file size, and the average number of input values per generated unit test. The table shows that the number of unit tests and the accompanying size of test code that the approach can produce are significant. The results confirm that we are able to provide an interface to verification tools via witnesses and tests that avoids technology lock-in and which enables developers to explore the verification results using tools and techniques they are familiar with. The combination of software verification and execution-based result validation may also be used to automatically extend the existing test suites of a project.

4.5 Detailed Discussion of Synthetic Examples

Now we discuss a few effects in more detail on hand-crafted example witnesses. Bugs that occur after only few loop iterations are also known as *shallow* bugs, as opposed to *deep* bugs that occur after many loop iterations. One of the strengths of dynamic validation approaches is that long loops can simply be executed, while model checkers usually need to perform expensive symbolic unrolling to reveal deep bugs, which is therefore a more difficult task for them than discovering shallow bugs. Thus, we expect the set of witnesses obtained from model checkers to consist mostly of shallow bugs, while at the same time we must expect that the advantages of test-based validation become most apparent for witnesses for deeper bugs, which necessitate many unrollings. Therefore, we hand-crafted a small set of verification tasks and witnesses, including the example for computing the mean from Fig. 1a in the introduction, to exemplify the differences between the test-based approaches and those based on model checking.

Figure 8a shows an example program intended to compare the iterative sum of ascending values with the result of the Gauss sum formula, and a witness for a bug in the program. The bug is located in lines 10 to 12 and causes an error for inputs larger than or equal to 10 000. The depicted witness for this bug assigns an input value of 10 000. Figure 8b shows an example program that increments two variables `x` and `y` 1 000 000 times and then asserts their equality in line 12, and a witness for a violation of this assertion. Since `y` is initialized to `x + 1` in line 5, the assertion will fail for any value of `x`. The depicted witness for this bug assigns an input value of 0. Figure 8c shows an example program with a variable `n` initialized with an input function in line 4 and copies its value to a variable `x` in line 5. In the same line, a variable `y` is initialized to 0. Then, in lines 6 to 9, `x` is decremented and simultaneously `y` is incremented, until `x` is 0, so essentially, `y` counts the loop iterations, and $n - x = y$ is a loop invariant. Consequently, `y` must be equal to `n` at the end of the loop, and therefore the call to



```c
1 extern void __VERIFIER_error(void);
2 extern unsigned int __VERIFIER_nondet_uint(void);
3 int main() {
4   unsigned int n = __VERIFIER_nondet_uint();
5   if (n < 1) return 0;
6   if (n > 1000000) return 0;
7   unsigned int sum = 0;
8   for (int i = 1; i <= n; i++) {
9     sum = sum + i;
10    if (i == 10000) {
11      sum = sum + 1;
12    }
13  }
14  if (2 * sum != n * (n + 1)) {
15    __VERIFIER_error();
16  }
17  return 0;
18 }
```

(a) "gauss" code, witness

```c
1 extern void __VERIFIER_error(void);
2 extern unsigned int __VERIFIER_nondet_uint(void);
3 int main(void) {
4   unsigned int x = __VERIFIER_nondet_uint();
5   unsigned int y = x + 1;
6   unsigned int i = 0;
7   while (i < 1000000) {
8     x++;
9     y++;
10    i++;
11  }
12  if (x != y) {
13    __VERIFIER_error();
14  }
15  return 0;
16 }
```

(b) "loop-1" code, witness

```c
1 extern void __VERIFIER_error();
2 extern unsigned int __VERIFIER_nondet_uint(void);
3 int main() {
4   unsigned int n = __VERIFIER_nondet_uint();
5   unsigned int x=n, y=0;
6   while (x > 0) {
7     x--;
8     y++;
9   }
10  if (y == n) {
11    __VERIFIER_error();
12  }
13  return 0;
14 }
```

(c) "loop-2" code, witnesses

Fig. 8. Hand-crafted tasks and witnesses

the error function in line 11 is called for any input value, so that both witnesses in Fig. 8c are valid counterexamples. The first of these witnesses, however, describes a violation that skips the loop entirely with an input value of 0, while the second one, due to assigning an input value of 1 000 000, reaches the violation in line 11 only after 1 000 000 loop iterations. We expect all validators to quickly validate the witnesses for shallow bugs, i.e., the one depicted in Fig. 1a and the first witness in Fig. 8c, but we expect test-based validators to perform significantly better on the witnesses for deep bugs, i.e., those depicted in Fig. 8a and 8b, and the second witness in Fig. 8c. Table 4 reports the results for validating these tasks and largely confirms our expectations. While CPAchecker exceeds its resource limitations ("M" for exceeding the memory limit, "T" for exceeding the CPU time limit) for all witnesses except for the two that represent shallow bugs, CPA-w2t and FShell-w2t quickly confirm all witnesses (✓). It is somewhat surprising to see that Ultimate Automizer is able to confirm the `loop-2/wit-2` of Fig. 8c. Checking the tool output, however, reveals that Ultimate Automizer ignored the input value of `n` specified by the witness and used 0 instead of 1 000 000. We were also surprised that the witnesses in the first two rows were rejected by Ultimate Automizer (✗), but since the confirmations of the execution-based validators along with their trustworthy executable tests give us confidence that the witnesses are correct, we assume that the rejections are either caused by the complexity of validating the witnesses or by an approximating behavior of Ultimate Automizer similar to the one leading to the rejection of `loop-2/wit-2`. Overall, we confirm that for this class of witnesses, dynamic approaches are more efficient and more effective than static approaches.

Table 4. Validation of hand-crafted witnesses

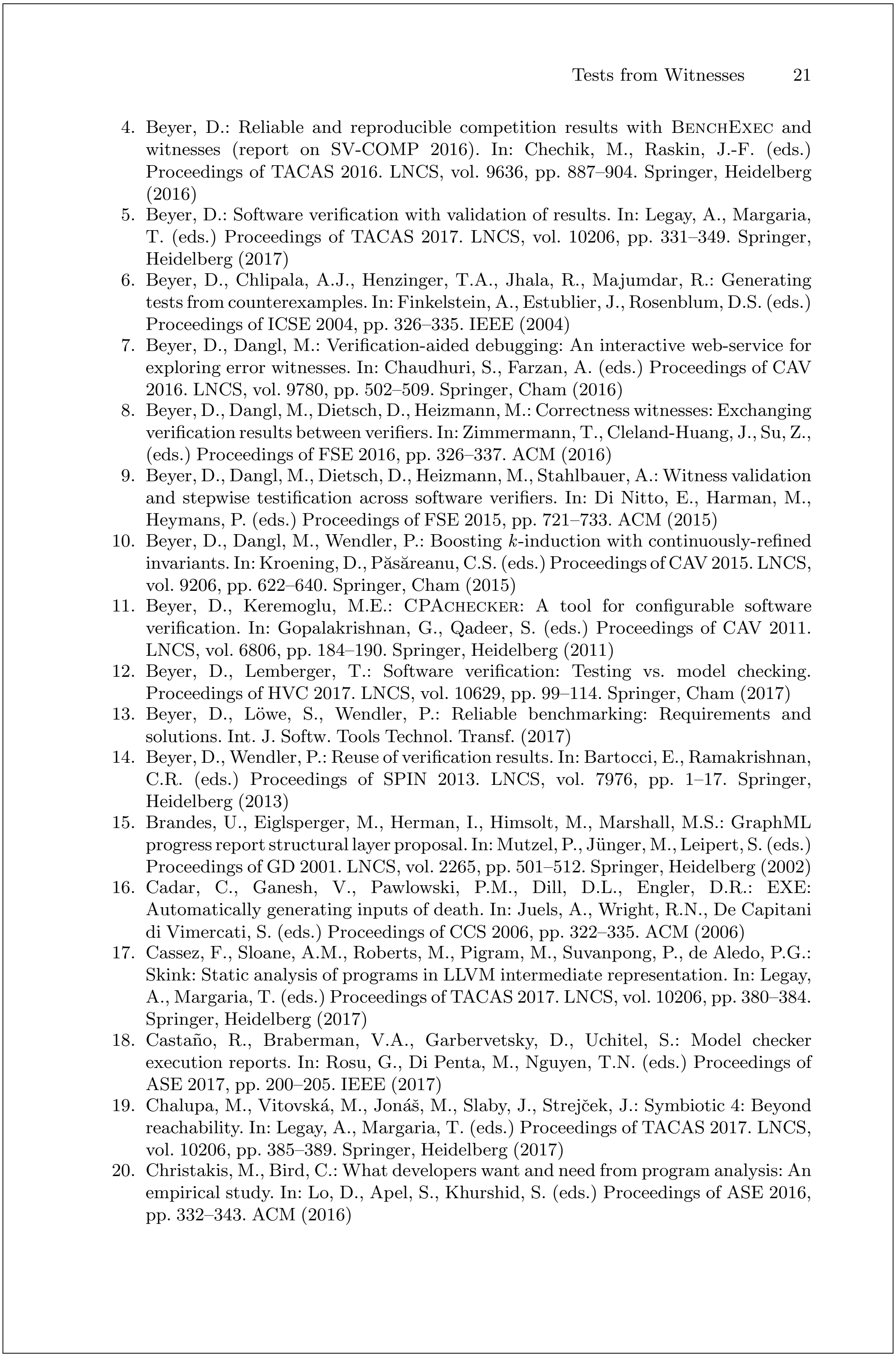
Witness	CPAchecker		Automizer		CPA-w2t		FShell-w2t	
	Result	Time (s)	Result	Time (s)	Result	Time (s)	Result	Time (s)
gauss	M	-	✗	11	✓	3.4	✓	0.60
loop-1	T	-	✗	9.6	✓	3.4	✓	0.60
loop-2/wit-1	✓	3.8	✓	8.0	✓	3.4	✓	0.58
loop-2/wit-2	T	-	✓	7.5	✓	3.2	✓	0.58
mean	✓	3.5	✓	7.1	✓	3.6	✓	0.58

5 Conclusion

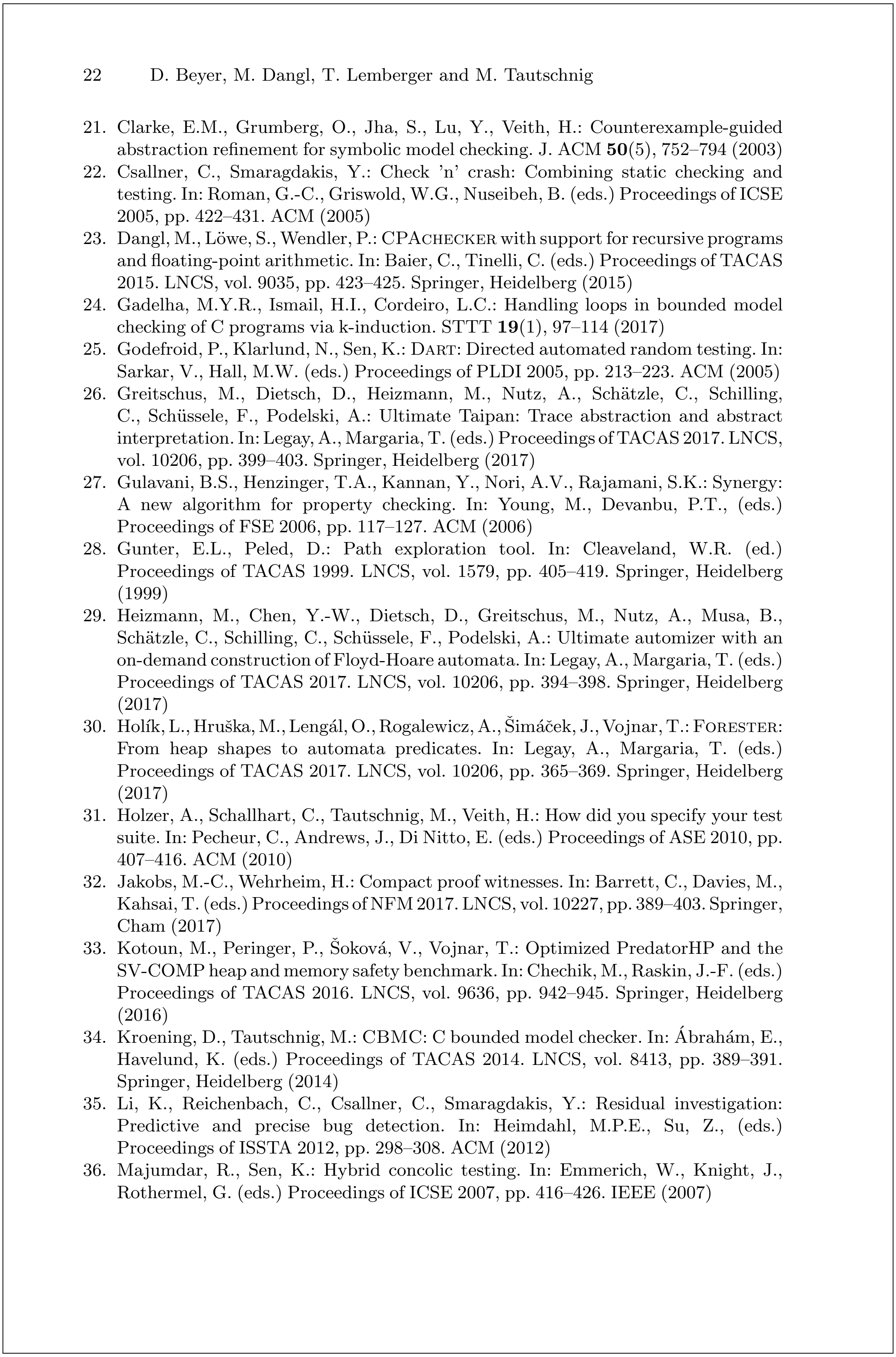
Developers are familiar with testing, and there are many tools available for bug analysis that are based on execution, such as debuggers. We try to close the gap between available verification tools and the desire for more precise bug finding by leveraging verification witnesses in an exchangeable standard format. We synthesize tests (test code) from verification results (witnesses) and check the tests for realizability by compiling them, linking them together with the original program, and executing the result in an isolating container. Prior to our work, developers would execute a verification tool and obtain the verification results, which include a violation witness in case a bug is found. Now, we can use the violation witness to obtain a test that drives the program to the specification violation (i.e., into the crash that the developer wants to investigate), while at the same time, we avoid verification-tool lock-in due to the exchangeable standard format. The approach reports only those tests to the developer that really expose the bug; any false alarms are suppressed. The results of our thorough experimental study are encouraging: We verified thousands of programs from the largest publicly-available collection of C verification tasks, consisting of 73 million lines of source code (2.3 GB), and synthesized tests that confirmed 7 286 verification results exposing known bugs in 974 different verification tasks.

References

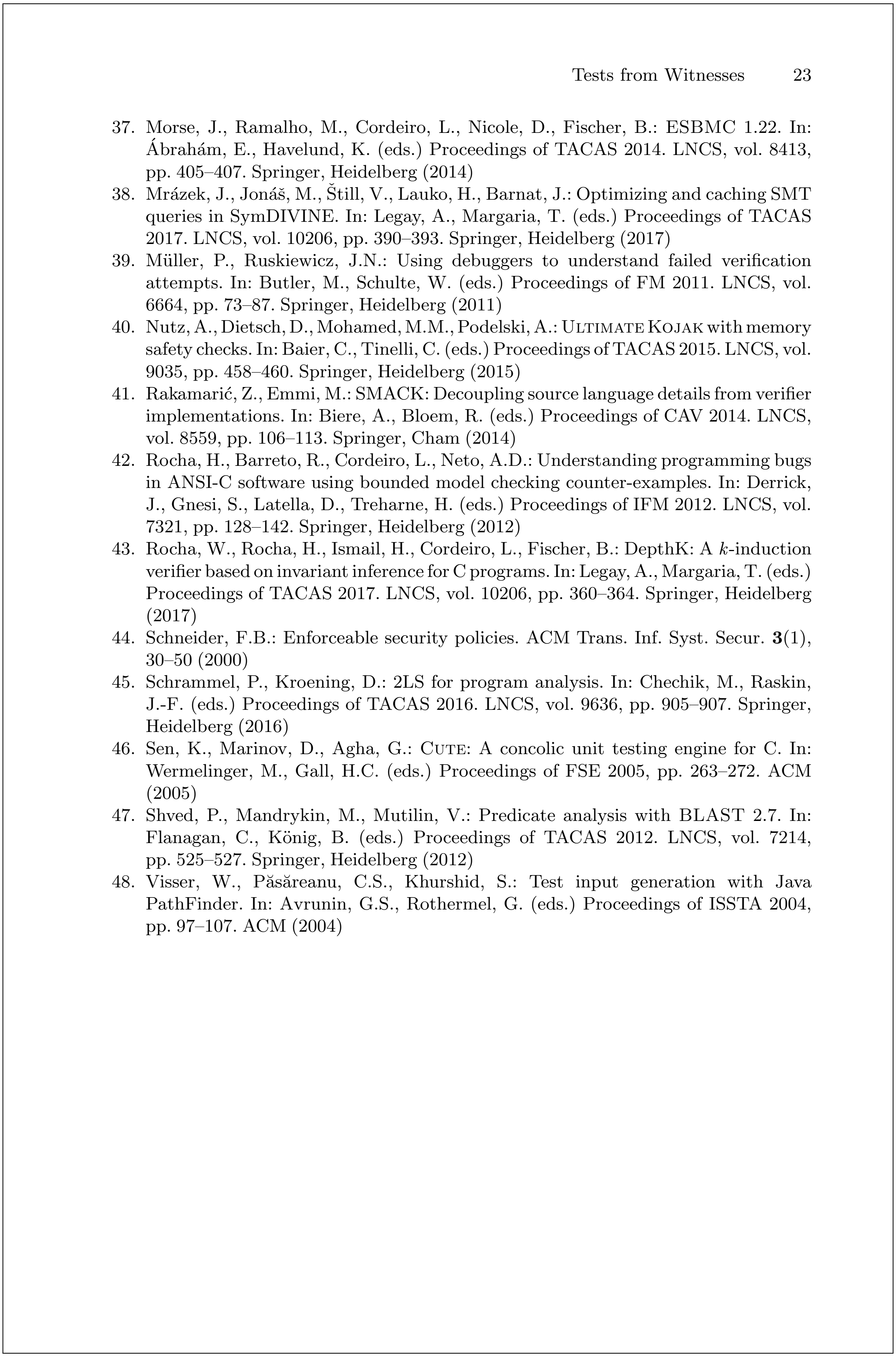
1. Alglave, J., Donaldson, A.F., Kroening, D., Tautschnig, M.: Making software verification tools really work. In: Bultan, T., Hsiung, P.-A. (eds.) Proceedings of ATVA 2011. LNCS, vol. 6996, pp. 28–42. Springer, Heidelberg (2011)
2. Andrianov, P., Friedberger, K., Mandrykin, M., Mutilin, V., Volkov, A.: CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 355–359. Springer, Heidelberg (2017)
3. Artho, C., Havelund, K., Honiden, S.: Visualization of concurrent program executions. In: Belli, F., Chen, A., Lin, H., McMillin, B., Mei, H. (eds.) Proceedings of COMPSAC 2007, pp. 541–546. IEEE (2007)



4. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In: Chechik, M., Raskin, J.-F. (eds.) Proceedings of TACAS 2016. LNCS, vol. 9636, pp. 887–904. Springer, Heidelberg (2016)
5. Beyer, D.: Software verification with validation of results. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017)
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) Proceedings of ICSE 2004, pp. 326–335. IEEE (2004)
7. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Chaudhuri, S., Farzan, A. (eds.) Proceedings of CAV 2016. LNCS, vol. 9780, pp. 502–509. Springer, Cham (2016)
8. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Zimmermann, T., Cleland-Huang, J., Su, Z., (eds.) Proceedings of FSE 2016, pp. 326–337. ACM (2016)
9. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Di Nitto, E., Harman, M., Heymans, P. (eds.) Proceedings of FSE 2015, pp. 721–733. ACM (2015)
10. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) Proceedings of CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Cham (2015)
11. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
12. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. Proceedings of HVC 2017. LNCS, vol. 10629, pp. 99–114. Springer, Cham (2017)
13. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transf. (2017)
14. Beyer, D., Wendler, P.: Reuse of verification results. In: Bartocci, E., Ramakrishnan, C.R. (eds.) Proceedings of SPIN 2013. LNCS, vol. 7976, pp. 1–17. Springer, Heidelberg (2013)
15. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report structural layer proposal. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) Proceedings of GD 2001. LNCS, vol. 2265, pp. 501–512. Springer, Heidelberg (2002)
16. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. In: Juels, A., Wright, R.N., De Capitani di Vimercati, S. (eds.) Proceedings of CCS 2006, pp. 322–335. ACM (2006)
17. Cassez, F., Sloane, A.M., Roberts, M., Pigram, M., Suvanpong, P., de Aledo, P.G.: Skink: Static analysis of programs in LLVM intermediate representation. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 380–384. Springer, Heidelberg (2017)
18. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Rosu, G., Di Penta, M., Nguyen, T.N. (eds.) Proceedings of ASE 2017, pp. 200–205. IEEE (2017)
19. Chalupa, M., Vitovská, M., Jonáš, M., Slaby, J., Strejček, J.: Symbiotic 4: Beyond reachability. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 385–389. Springer, Heidelberg (2017)
20. Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of ASE 2016, pp. 332–343. ACM (2016)



21. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003)
22. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Roman, G.-C., Griswold, W.G., Nuseibeh, B. (eds.) Proceedings of ICSE 2005, pp. 422–431. ACM (2005)
23. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic. In: Baier, C., Tinelli, C. (eds.) Proceedings of TACAS 2015. LNCS, vol. 9035, pp. 423–425. Springer, Heidelberg (2015)
24. Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. STTT **19**(1), 97–114 (2017)
25. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of PLDI 2005, pp. 213–223. ACM (2005)
26. Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Taipan: Trace abstraction and abstract interpretation. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 399–403. Springer, Heidelberg (2017)
27. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Young, M., Devanbu, P.T., (eds.) Proceedings of FSE 2006, pp. 117–127. ACM (2006)
28. Gunter, E.L., Peled, D.: Path exploration tool. In: Cleaveland, W.R. (ed.) Proceedings of TACAS 1999. LNCS, vol. 1579, pp. 405–419. Springer, Heidelberg (1999)
29. Heizmann, M., Chen, Y.-W., Dietsch, D., Greitschus, M., Nutz, A., Musa, B., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate automizer with an on-demand construction of Floyd-Hoare automata. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 394–398. Springer, Heidelberg (2017)
30. Holík, L., Hruška, M., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forester: From heap shapes to automata predicates. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 365–369. Springer, Heidelberg (2017)
31. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Pecheur, C., Andrews, J., Di Nitto, E. (eds.) Proceedings of ASE 2010, pp. 407–416. ACM (2010)
32. Jakobs, M.-C., Wehrheim, H.: Compact proof witnesses. In: Barrett, C., Davies, M., Kahsai, T. (eds.) Proceedings of NFM 2017. LNCS, vol. 10227, pp. 389–403. Springer, Cham (2017)
33. Kotoun, M., Peringer, P., Šoková, V., Vojnar, T.: Optimized PredatorHP and the SV-COMP heap and memory safety benchmark. In: Chechik, M., Raskin, J.-F. (eds.) Proceedings of TACAS 2016. LNCS, vol. 9636, pp. 942–945. Springer, Heidelberg (2016)
34. Kroening, D., Tautschnig, M.: CBMC: C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) Proceedings of TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
35. Li, K., Reichenbach, C., Csallner, C., Smaragdakis, Y.: Residual investigation: Predictive and precise bug detection. In: Heimdahl, M.P.E., Su, Z., (eds.) Proceedings of ISSTA 2012, pp. 298–308. ACM (2012)
36. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Emmerich, W., Knight, J., Rothermel, G. (eds.) Proceedings of ICSE 2007, pp. 416–426. IEEE (2007)



37. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: ESBMC 1.22. In: Ábrahám, E., Havelund, K. (eds.) Proceedings of TACAS 2014. LNCS, vol. 8413, pp. 405–407. Springer, Heidelberg (2014)
38. Mrázek, J., Jonáš, M., Štill, V., Lauko, H., Barnat, J.: Optimizing and caching SMT queries in SymDIVINE. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 390–393. Springer, Heidelberg (2017)
39. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Butler, M., Schulte, W. (eds.) Proceedings of FM 2011. LNCS, vol. 6664, pp. 73–87. Springer, Heidelberg (2011)
40. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: Ultimate Kojak with memory safety checks. In: Baier, C., Tinelli, C. (eds.) Proceedings of TACAS 2015. LNCS, vol. 9035, pp. 458–460. Springer, Heidelberg (2015)
41. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) Proceedings of CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014)
42. Rocha, H., Barreto, R., Cordeiro, L., Neto, A.D.: Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) Proceedings of IFM 2012. LNCS, vol. 7321, pp. 128–142. Springer, Heidelberg (2012)
43. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., Fischer, B.: DepthK: A k-induction verifier based on invariant inference for C programs. In: Legay, A., Margaria, T. (eds.) Proceedings of TACAS 2017. LNCS, vol. 10206, pp. 360–364. Springer, Heidelberg (2017)
44. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000)
45. Schrammel, P., Kroening, D.: 2LS for program analysis. In: Chechik, M., Raskin, J.-F. (eds.) Proceedings of TACAS 2016. LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016)
46. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for C. In: Wermelinger, M., Gall, H.C. (eds.) Proceedings of FSE 2005, pp. 263–272. ACM (2005)
47. Shved, P., Mandrykin, M., Mutilin, V.: Predicate analysis with BLAST 2.7. In: Flanagan, C., König, B. (eds.) Proceedings of TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
48. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Avrunin, G.S., Rothermel, G. (eds.) Proceedings of ISSTA 2004, pp. 97–107. ACM (2004)

Proc. STTT 2023, © Springer

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Six Years Later: Testing vs. Model Checking

Dirk Beyer and Thomas Lemberger

LMU Munich, Germany

The date of receipt and acceptance will be inserted by the editor

Abstract. Six years ago, we performed the first large-scale comparison of automated test generators and software model checkers with respect to bug-finding capabilities on a benchmark set with 5 693 C programs. Since then, the International Competition on Software Testing (Test-Comp) has established standardized formats and community-agreed rules for the experimental comparison of test generators. With this new context, it is time to revisit our initial question: Model checkers or test generators—which tools are more effective in finding bugs in software? To answer this, we perform a comparative analysis on the tools and existing data published by two competitions, the International Competition on Software Verification (SV-COMP) and Test-Comp. The results provide two insights: (1) Almost all test generators that participate in Test-Comp use hybrid approaches that include formal methods, and (2) while the considered model checkers are still highly competitive, the considered test generators' bug-finding capabilities now outperform them.

Key words: Software verification, Model checking, Program analysis, Test-case generation, Testing, Fuzzing

1 Introduction

In previous research [26] we compare the bug-finding capabilities of automated test generators and software model checkers on C programs. At the time of this work, no standardized formats existed for the experimental comparison of test generators, so we had to manually implement adapters for a selection of off-the-shelf test generators and our own coverage measurement. Nowadays, the International Competition on Software Testing (Test-Comp) [16] provides a community-set framework for the evaluation of test generators for the C language, including an exchange format for test suites, a large and well-defined benchmark task set, and agreed-upon resource limitations for benchmarking. So far, the benchmark test tasks of Test-Comp target two goals of test generation: "create a test suite that covers a known bug in a given program", and "create a test suite that covers as many as possible branches of a given program".

Thanks to the improvements Test-Comp brought, and six years after our original research [26], it is time to revisit the comparison: Model checkers vs. test generators—which tools are better at finding bugs in software?

We improve on the original comparison in multiple ways: (1) For the original work, we selected an array of test generators manually, and configured them to the best of our knowledge. In this work, we base our comparison only on participants of the International Competition on Software Verification (SV-COMP) [13] and Test-Comp. All tool configuration is provided by the participating tool developers, and during the competition, developers got early access to pre-run results to fix any shortcomings of their tools evident through the benchmark set.

(2) Previously, we executed our own, novel experiments. We do have high confidence in these results, but in this work, we reuse the freely available competition data of SV-COMP 2023 and Test-Comp 2023. Using these results has the advantage that the data was peer-reviewed by the tool developers before publication.

Through these two adjustments we ensure that the used experimental data represents expert tool usage. It also guarantees that we configured everything correctly, and that we select tools that support all of the major required language features.

(3) Originally, we said that a model checker found a bug when it reports a bug and this report is confirmed by at least one witness validator [20]. In this work, we pay higher tribute to the actual execution of an error. We

separately consider whether a model checker's bug report can be validated through program execution [21].

(4) In the original work, we only consider the bug-finding capabilities of model checkers and test generators, but do not explicitly tune test generators towards finding a bug in the program. Our expectation is that many test generators are originally designed for traditional coverage measures like branch coverage or condition coverage, and are not optimized to create a single test for an error location of interest. But since Test-Comp asks participants to create a test suite that covers a known bug, the Test-Comp test generators may be tuned towards bug-finding. To check the effect of this, we compare the test suites generated by Test-Comp test generators for error coverage and the test suites generated for branch coverage with regards to their bug-finding capabilities.

(5) Furthermore, in the original work we compare tools that market themselves as software model checkers with tools that market themselves as test generators, and give a coarse overview on the techniques they use. Nowadays, many tools employ hybrid approaches with multiple different techniques. Many formal methods that are used in model checking can also be used for test generation [18], and techniques originally designed for testing can be used as a part of model checking, for example input fuzzing [44]. This means that a model checker and a test generator may use the same underlying analysis techniques. In this work, we consider the individual analysis techniques in detail.

In summary, we pose the following questions:

RQ 1 Are test generators more effective in finding bugs than software model checkers?
RQ 2 Can bug reports of software model checkers be validated through execution?
RQ 3 Are test generators that target errors more effective in finding bugs than test generators that target branch coverage?

To answer these questions, we use Test-Comp test generators and SV-COMP model checkers as representatives of their respective domains, with the original competition data. To the best of our knowledge, this is the first meta-analysis of the two international competitions SV-COMP and Test-Comp, and the largest evaluation that compares the bug-finding capabilities of software model checkers with those of test generators.

Related Work. The only large-scale comparisons of the tools considered in this work are the annual competitions SV-COMP [13] and Test-Comp [16], which we combine and inspect in detail in this work.

While there are no other large-scale comparisons of tools, there are literature surveys on test generation for JavaScript [6], search-based testing [78], fuzzing [77], and symbolic execution [8, 38, 84]. There are also surveys on software-model-checking techniques [52, 68] and formal methods in a more general sense [11, 57], as well as the handbook on model checking [45].

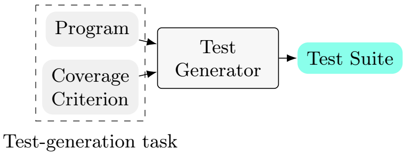


Fig. 1: Workflow of a Test-Comp test generator. A test generator produces a test suite for a program under test and a coverage criterion.

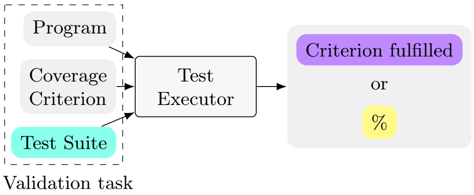


Fig. 2: Workflow of a test executor. A test executor computes whether (or to what percentage) a test suite fulfills a coverage criterion for a program.

This work focuses on reachability bugs in a sequential, self-sufficient program, similar to a failing `assert` statement, and on tools and techniques aimed at finding such errors. Other domains of model checking and automated testing are, among many others, protocol verification [10], grammar-based testing [32, 58], and mutation testing [83].

2 Background

2.1 Testing

An input function in a program is any function that retrieves a value from the program's environment, for example a system call. In our work, we use special functions `__VERIFIER_nondet_X` that can return any input value of type `X`. For example, function `__VERIFIER_nondet_int()` returns an integer input value. A *test* $\langle v_0, \dots, v_n \rangle$ is a sequence of n values. When $\langle v_0, \dots, v_n \rangle$ is executed, the i-th call to an input function is defined to return value v_i. A test suite is a set of tests.

A test t covers a program operation *op* if the execution of t goes through *op*. A test suite covers a program operation *op* if any of its contained tests covers *op*.

A Test-Comp test generator (Fig. 1) [16] takes as input the program-under-test and a coverage-criterion (e.g., cover a call to function `reach_error()`), and generates as output a test suite.

The test executor (Fig. 2) then takes as input the program-under-test, the coverage criterion, and the generated test suite. It produces as output either that the

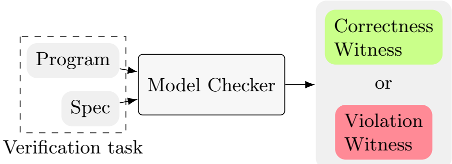


Fig. 3: Workflow of a model checker. A model checker produces a correctness witness when it claims that the program under verification fulfills the specification, or a violation witness when it claims that the program violates the specification.

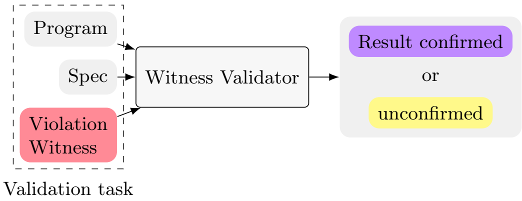


Fig. 4: Workflow of verification-result validation for violation witnesses with a witness validator. A witness validator confirms the model checker's verification result if it can reproduce the result with the help of the witness.

coverage criterion is fulfilled, or a percentage of how many coverage goals that are defined by the criterion are covered by the tests in the test suite.

2.2 Model Checking

A SV-COMP model checker (Fig. 3) [13] takes as input a program and a specification and produces one of two outputs: If the program fulfills the specification, a correctness witness [19] is generated. If the program violates the specification, a violation witness [19] is generated.

2.3 Witness Validation

Witness validation [19] aims to increase the trust in results of model checking. The idea is the following: A model checker (Fig. 3) analyzes a program with regards to a specification. As output, it not only produces a verification verdict *"property fulfilled"* or *"property not fulfilled"*, but also a correctness witness or violation witness that helps to recreate the verification result. This witness is then given to a *witness validator* (Fig. 4). A witness validator takes the program-under-verification, the original specification, and the previously produced witness as input. It tries to reproduce the verification result with the help of the witness. If the witness validator is successful, the result is confirmed and confidence in the verification result increases.

```
1  unsigned char __VERIFIER_nondet_uchar();
2  void reach_error();
3
4  int main() {
5    unsigned char a =
6      __VERIFIER_nondet_uchar();
7    unsigned char b =
8      __VERIFIER_nondet_uchar();
9    unsigned char sum = a + b;
10   unsigned char mean = sum / 2;
11   if (mean < a / 2) {
12     reach_error();
13   }
14 }
```

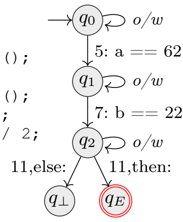


Fig. 5: Example program and violation-witness automaton (adapted from prior work [21])

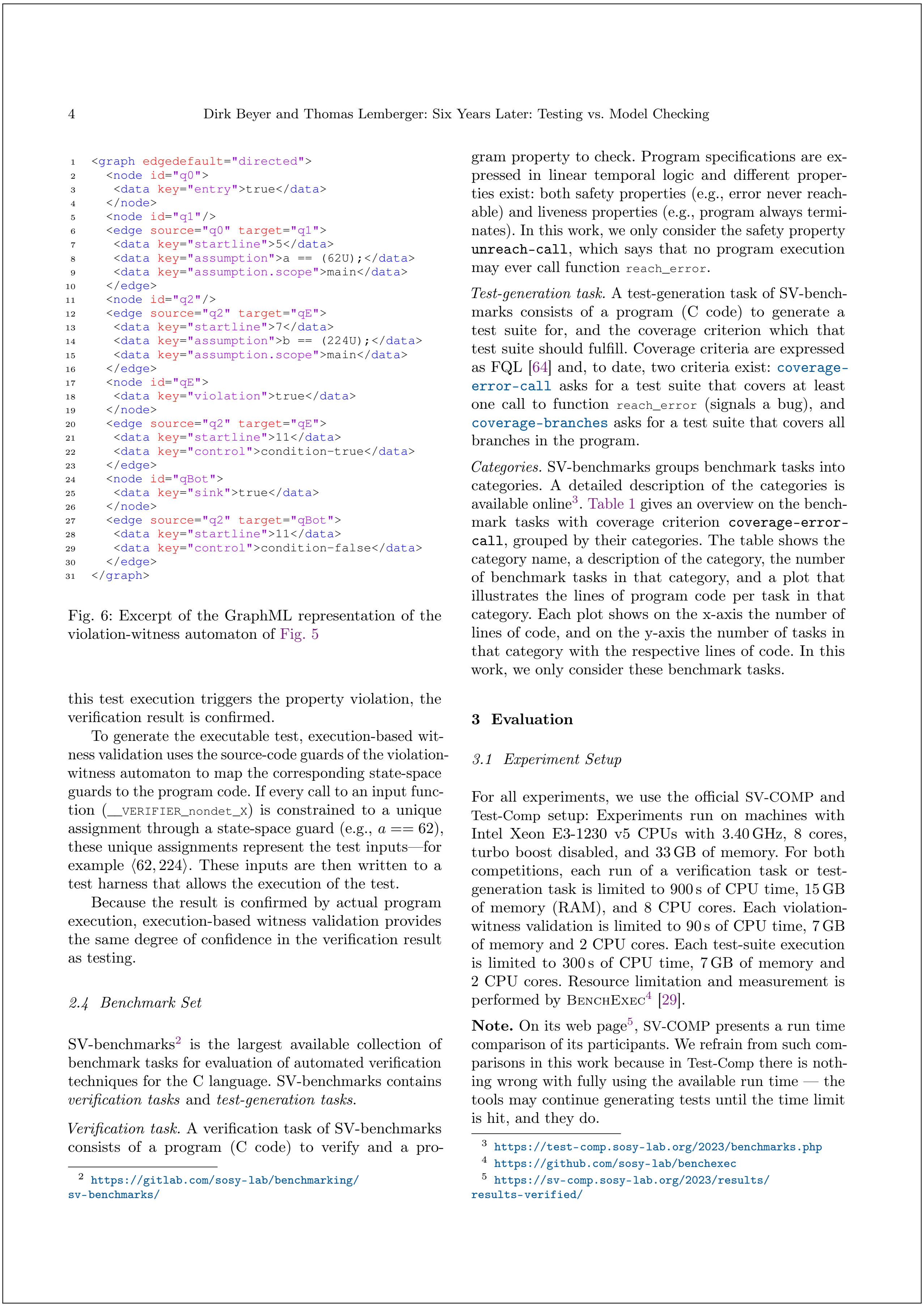
In this work, we focus on bug-finding capabilities, so we only consider violation witnesses.

We describe violation witnesses as *violation-witness automata*. A violation-witness automaton is a finite-state automaton. It contains at its transitions *source-code guards* e and *state-space guards* ψ to describe a subset of the program state space that contains the reported property violation. A source-code guard e is a program statement identified by its source-code line number. A source-code guard can also restrict the direction of program branchings, for example at if-statements. It only allows the transition from one witness-automaton state to another if the currently considered program expression matches e and the specified program branch is entered (if specified). A state-space guard ψ is a predicate on the program state. It restricts the possible program states to those that fulfill ψ. Figure 5 shows an example program and a violation-witness automaton for the violated property `unreach-call`. Automaton label o/w describes a transition that is taken in all cases not covered by other transitions. This violation-witness automaton describes only the program state space that assigns `a = 62` and `b = 224`, which leads to an unsigned integer overflow and makes the program enter the if-branch: The automaton stays in state q_0 until the assignment in line 5 is considered. It then transitions to q_1 and restricts the considered program states to those that fulfill $a == 62$ (after transitioning). When line 7 is reached, it restricts the considered program states to those that fulfill $b == 224$. When the if-statement in line 11 is reached and the if-branch is entered, the violation is reached.

SV-COMP requires participants to output violation witnesses since SV-COMP 2015 [12]. It uses the XML-based GraphML exchange format[1]. Figure 6 shows an excerpt that represents the automaton displayed in Fig. 5.

Witness to Test. Execution-based witness validation [21] takes a violation witness and tries to transform it into an executable test. If it succeeds, the test is executed. If

[1] https://github.com/sosy-lab/sv-witnesses



```
<graph edgedefault="directed">
  <node id="q0">
   <data key="entry">true</data>
  </node>
  <node id="q1"/>
  <edge source="q0" target="q1">
   <data key="startline">5</data>
   <data key="assumption">a == (62U);</data>
   <data key="assumption.scope">main</data>
  </edge>
  <node id="q2"/>
  <edge source="q2" target="qE">
   <data key="startline">7</data>
   <data key="assumption">b == (224U);</data>
   <data key="assumption.scope">main</data>
  </edge>
  <node id="qE">
   <data key="violation">true</data>
  </node>
  <edge source="q2" target="qE">
   <data key="startline">11</data>
   <data key="control">condition-true</data>
  </edge>
  <node id="qBot">
   <data key="sink">true</data>
  </node>
  <edge source="q2" target="qBot">
   <data key="startline">11</data>
   <data key="control">condition-false</data>
  </edge>
</graph>
```

Fig. 6: Excerpt of the GraphML representation of the violation-witness automaton of Fig. 5

this test execution triggers the property violation, the verification result is confirmed.

To generate the executable test, execution-based witness validation uses the source-code guards of the violation-witness automaton to map the corresponding state-space guards to the program code. If every call to an input function (__VERIFIER_nondet_X) is constrained to a unique assignment through a state-space guard (e.g., $a == 62$), these unique assignments represent the test inputs—for example $\langle 62, 224 \rangle$. These inputs are then written to a test harness that allows the execution of the test.

Because the result is confirmed by actual program execution, execution-based witness validation provides the same degree of confidence in the verification result as testing.

2.4 Benchmark Set

SV-benchmarks[2] is the largest available collection of benchmark tasks for evaluation of automated verification techniques for the C language. SV-benchmarks contains *verification tasks* and *test-generation tasks.*

Verification task. A verification task of SV-benchmarks consists of a program (C code) to verify and a program property to check. Program specifications are expressed in linear temporal logic and different properties exist: both safety properties (e.g., error never reachable) and liveness properties (e.g., program always terminates). In this work, we only consider the safety property **unreach-call**, which says that no program execution may ever call function reach_error.

Test-generation task. A test-generation task of SV-benchmarks consists of a program (C code) to generate a test suite for, and the coverage criterion which that test suite should fulfill. Coverage criteria are expressed as FQL [64] and, to date, two criteria exist: coverage-error-call asks for a test suite that covers at least one call to function reach_error (signals a bug), and coverage-branches asks for a test suite that covers all branches in the program.

Categories. SV-benchmarks groups benchmark tasks into categories. A detailed description of the categories is available online[3]. Table 1 gives an overview on the benchmark tasks with coverage criterion **coverage-error-call**, grouped by their categories. The table shows the category name, a description of the category, the number of benchmark tasks in that category, and a plot that illustrates the lines of program code per task in that category. Each plot shows on the x-axis the number of lines of code, and on the y-axis the number of tasks in that category with the respective lines of code. In this work, we only consider these benchmark tasks.

3 Evaluation

3.1 Experiment Setup

For all experiments, we use the official SV-COMP and Test-Comp setup: Experiments run on machines with Intel Xeon E3-1230 v5 CPUs with 3.40 GHz, 8 cores, turbo boost disabled, and 33 GB of memory. For both competitions, each run of a verification task or test-generation task is limited to 900 s of CPU time, 15 GB of memory (RAM), and 8 CPU cores. Each violation-witness validation is limited to 90 s of CPU time, 7 GB of memory and 2 CPU cores. Each test-suite execution is limited to 300 s of CPU time, 7 GB of memory and 2 CPU cores. Resource limitation and measurement is performed by BenchExec[4] [29].

Note. On its web page[5], SV-COMP presents a run time comparison of its participants. We refrain from such comparisons in this work because in Test-Comp there is nothing wrong with fully using the available run time — the tools may continue generating tests until the time limit is hit, and they do.

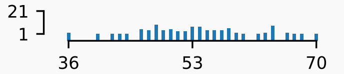
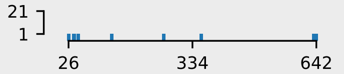
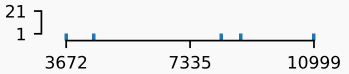
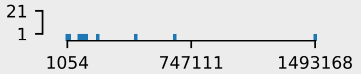
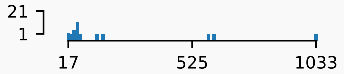
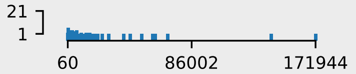
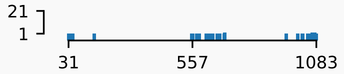
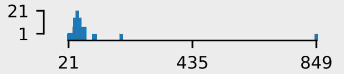
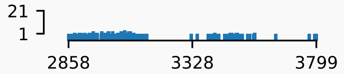
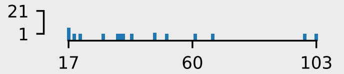
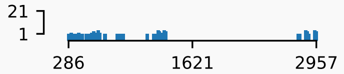
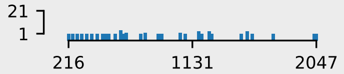
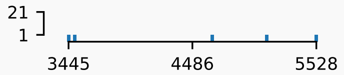
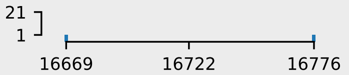
[2] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/

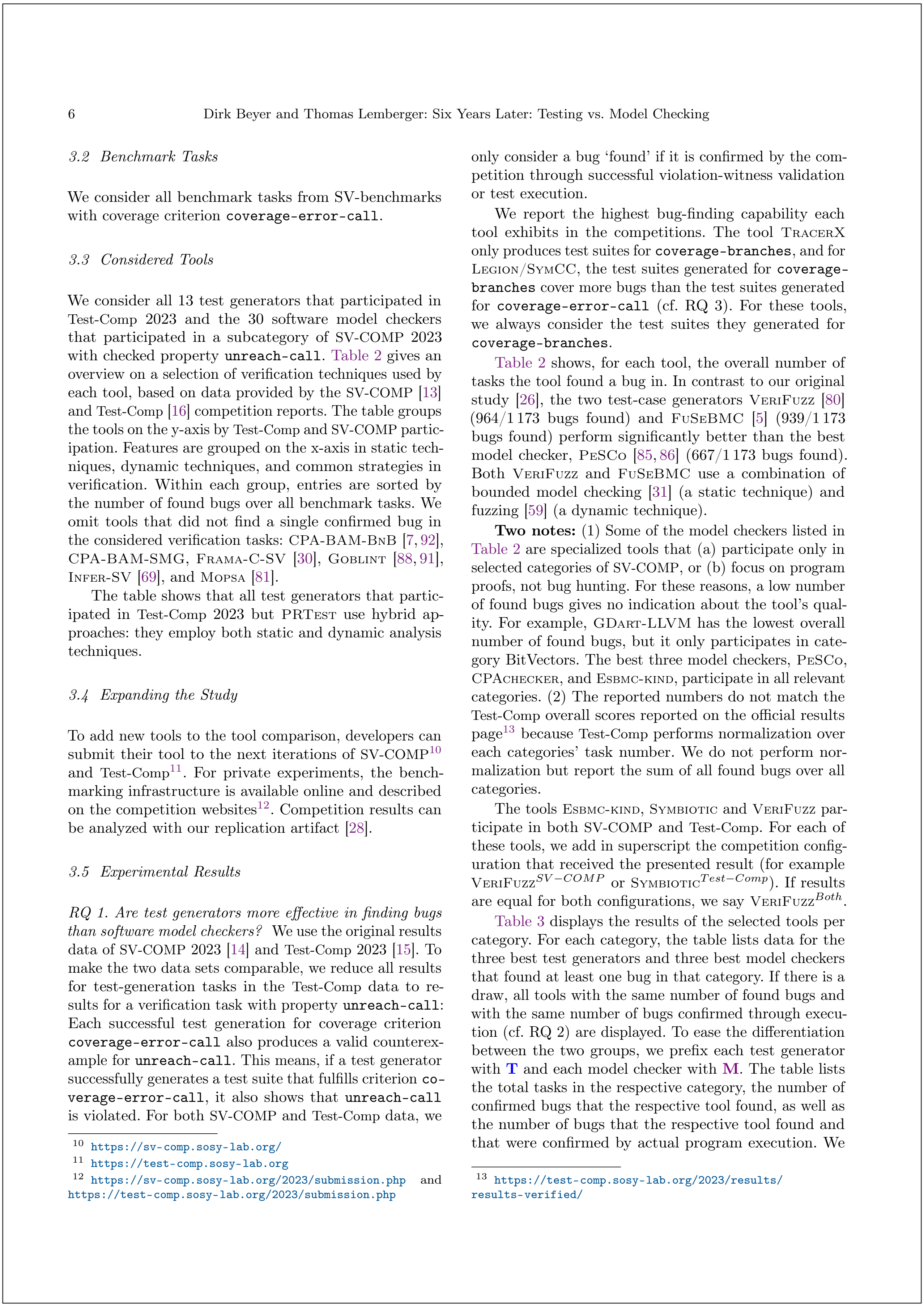
[3] https://test-comp.sosy-lab.org/2023/benchmarks.php

[4] https://github.com/sosy-lab/benchexec

[5] https://sv-comp.sosy-lab.org/2023/results/results-verified/

Table 1: Subcategories (13) of Test-Comp with coverage criterion `coverage-error-call`. Each plot in the column 'Lines of Code' illustrates the lines of program code per task in that category. Each plot shows on the x-axis the number of lines of code, and on the y-axis the number of tasks in that category with the respective lines of code.

Subcategory	Description	#Tasks	Lines of Code
Arrays	Require treatment of arrays	90	
BitVectors	Require treatment of bit-operations	9	
ControlFlow	Program correctness depends mostly on the control-flow structure and integer variables.	5	
ECA	Derived from event-condition-action systems	18	
Floats	Require treatment of floating-point arithmetics	32	
Hardware	Created from word-level hardware-model-checking benchmarks	494	
Heap	Require treatment of data structures on the heap, pointer aliases, and function pointers	47	
Loops	Require treatment of (potentially indeterminate) loops	130	
ProductLines	Represent 'products' and 'product simulators' that are derived using different configurations of product lines	169	
Recursive	Require treatment of recursive functions	20	
Sequentialized	Sequentialized concurrent programs that were derived from SystemC programs. The programs were transformed to pure C programs by incorporating the scheduler into the C code	98	
XCSP	Derived from constraint-programming benchmark tasks of combinatorial constrained problems	54	
BusyBox	Tasks from the software system BusyBox	5	
DeviceDriversLinux64	Tasks from the Linux Driver Verification project	2	



3.2 Benchmark Tasks

We consider all benchmark tasks from SV-benchmarks with coverage criterion `coverage-error-call`.

3.3 Considered Tools

We consider all 13 test generators that participated in Test-Comp 2023 and the 30 software model checkers that participated in a subcategory of SV-COMP 2023 with checked property `unreach-call`. Table 2 gives an overview on a selection of verification techniques used by each tool, based on data provided by the SV-COMP [13] and Test-Comp [16] competition reports. The table groups the tools on the y-axis by Test-Comp and SV-COMP participation. Features are grouped on the x-axis in static techniques, dynamic techniques, and common strategies in verification. Within each group, entries are sorted by the number of found bugs over all benchmark tasks. We omit tools that did not find a single confirmed bug in the considered verification tasks: CPA-BAM-BnB [7, 92], CPA-BAM-SMG, Frama-C-SV [30], Goblint [88, 91], Infer-SV [69], and Mopsa [81].

The table shows that all test generators that participated in Test-Comp 2023 but PRTest use hybrid approaches: they employ both static and dynamic analysis techniques.

3.4 Expanding the Study

To add new tools to the tool comparison, developers can submit their tool to the next iterations of SV-COMP[10] and Test-Comp[11]. For private experiments, the benchmarking infrastructure is available online and described on the competition websites[12]. Competition results can be analyzed with our replication artifact [28].

3.5 Experimental Results

RQ 1. Are test generators more effective in finding bugs than software model checkers? We use the original results data of SV-COMP 2023 [14] and Test-Comp 2023 [15]. To make the two data sets comparable, we reduce all results for test-generation tasks in the Test-Comp data to results for a verification task with property `unreach-call`: Each successful test generation for coverage criterion `coverage-error-call` also produces a valid counterexample for `unreach-call`. This means, if a test generator successfully generates a test suite that fulfills criterion `coverage-error-call`, it also shows that `unreach-call` is violated. For both SV-COMP and Test-Comp data, we only consider a bug 'found' if it is confirmed by the competition through successful violation-witness validation or test execution.

We report the highest bug-finding capability each tool exhibits in the competitions. The tool TracerX only produces test suites for `coverage-branches`, and for Legion/SymCC, the test suites generated for `coverage-branches` cover more bugs than the test suites generated for `coverage-error-call` (cf. RQ 3). For these tools, we always consider the test suites they generated for `coverage-branches`.

Table 2 shows, for each tool, the overall number of tasks the tool found a bug in. In contrast to our original study [26], the two test-case generators VeriFuzz [80] (964/1 173 bugs found) and FuSeBMC [5] (939/1 173 bugs found) perform significantly better than the best model checker, PeSCo [85, 86] (667/1 173 bugs found). Both VeriFuzz and FuSeBMC use a combination of bounded model checking [31] (a static technique) and fuzzing [59] (a dynamic technique).

Two notes: (1) Some of the model checkers listed in Table 2 are specialized tools that (a) participate only in selected categories of SV-COMP, or (b) focus on program proofs, not bug hunting. For these reasons, a low number of found bugs gives no indication about the tool's quality. For example, GDart-LLVM has the lowest overall number of found bugs, but it only participates in category BitVectors. The best three model checkers, PeSCo, CPAchecker, and Esbmc-kind, participate in all relevant categories. (2) The reported numbers do not match the Test-Comp overall scores reported on the official results page[13] because Test-Comp performs normalization over each categories' task number. We do not perform normalization but report the sum of all found bugs over all categories.

The tools Esbmc-kind, Symbiotic and VeriFuzz participate in both SV-COMP and Test-Comp. For each of these tools, we add in superscript the competition configuration that received the presented result (for example VeriFuzz$^{SV-COMP}$ or Symbiotic$^{Test-Comp}$). If results are equal for both configurations, we say VeriFuzz^{Both}.

Table 3 displays the results of the selected tools per category. For each category, the table lists data for the three best test generators and three best model checkers that found at least one bug in that category. If there is a draw, all tools with the same number of found bugs and with the same number of bugs confirmed through execution (cf. RQ 2) are displayed. To ease the differentiation between the two groups, we prefix each test generator with **T** and each model checker with **M**. The table lists the total tasks in the respective category, the number of confirmed bugs that the respective tool found, as well as the number of bugs that the respective tool found and that were confirmed by actual program execution. We

[10] https://sv-comp.sosy-lab.org/

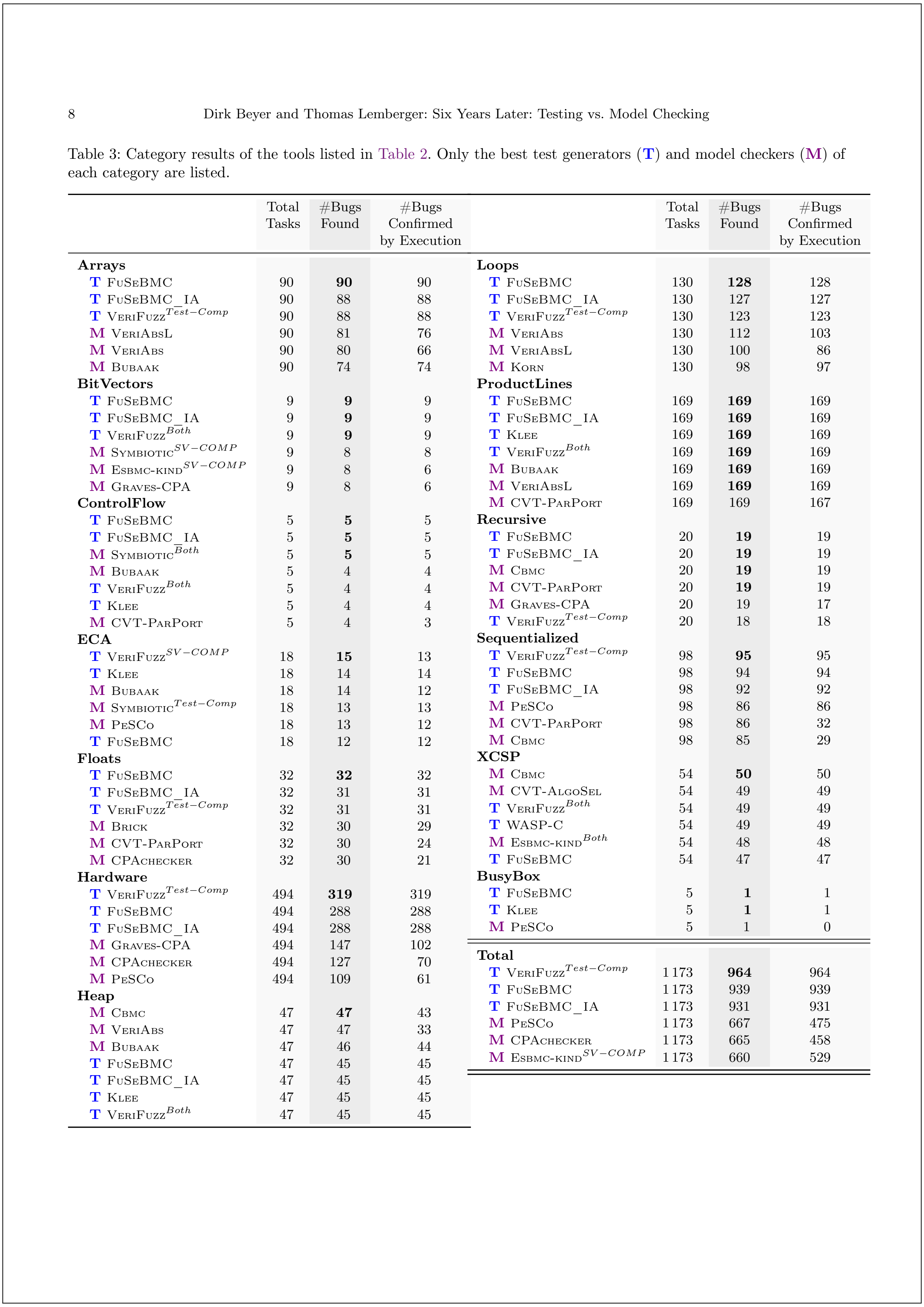
[11] https://test-comp.sosy-lab.org

[12] https://sv-comp.sosy-lab.org/2023/submission.php and https://test-comp.sosy-lab.org/2023/submission.php

[13] https://test-comp.sosy-lab.org/2023/results/results-verified/

Table 2: Features used by Test-Comp and SV-COMP participants and their overall results in bug finding

		Static								Dyn.		Strategies										
	Participant	**Bounded Model Checking**	**CEGAR**	**Explicit-Value Analysis**	**k-Induction**	**Numeric Interval Analysis**	**Predicate Abstraction**	**Shape Analysis**	**Symbolic Execution**	**Random Execution**	**Evolutionary Algorithms**	**ARG-Based Analysis**	**Bit-Precise Analysis**	**Floating-Point Arithmetics**	**Lazy Abstraction**	**Interpolation**	**Automata-Based Analysis**	**Guidance by Property**	**Targeted Input Generation**	**Algorithm Selection**	**Portfolio**	#Bugs Found
Test-Comp	VeriFuzz [79, 80]	✓		✓		✓				✓	✓			✓				✓				964
	FuSeBMC [4, 5]	✓				✓				✓	✓		✓	✓				✓	✓		✓	939
	FuSeBMC_IA [3]	✓								✓	✓		✓	✓				✓	✓		✓	931
	CoVeriTest [22, 67]		✓	✓			✓						✓	✓							✓	564
	Klee [36, 37]								✓				✓	✓					✓			541
	Symbiotic [40, 41]								✓				✓	✓				✓	✓		✓	510
	TracerX [65, 66]	✓							✓				✓	✓		✓			✓			420
	HybridTiger [35, 87]		✓	✓			✓						✓	✓								397
	WASP-C [6]								✓	✓				✓								393
	Esbmc-kind [55, 56]	✓			✓	✓							✓					✓				352
	PRTest [26, 73]									✓			✓	✓								293
	Legion/SymCC [7]			✓					✓	✓			✓	✓				✓	✓			281
	Legion [74, 75]			✓					✓	✓			✓	✓				✓	✓			108
SV-COMP	PeSCo [85, 86]	✓	✓	✓	✓	✓	✓	✓				✓	✓		✓	✓		✓		✓	✓	667
	CPAchecker [25, 47]	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓		✓		✓	✓	665
	Esbmc-kind [55, 56]	✓			✓	✓							✓					✓				660
	VeriAbsL [49]	✓	✓	✓	✓	✓	✓			✓	✓							✓		✓	✓	645
	Graves-CPA [72]																					643
	VeriAbs [2, 48]	✓	✓	✓	✓	✓	✓			✓	✓							✓		✓	✓	639
	Bubaak [39]								✓				✓					✓				635
	Cbmc [46, 70]	✓											✓					✓				626
	VeriFuzz [44, 79]	✓				✓				✓	✓							✓				615
	CVT-ParPort [23, 24]	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓		✓		✓	✓	591
	Symbiotic [41, 42]				✓	✓		✓	✓				✓					✓			✓	559
	CVT-AlgoSel [23, 24]	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓		✓	✓	468
	UAutomizer [62, 63]		✓				✓						✓		✓	✓	✓	✓		✓	✓	311
	Divine [9, 71]			✓					✓				✓					✓		✓	✓	299
	UTaipan [50, 60]		✓	✓		✓	✓						✓		✓	✓	✓	✓		✓	✓	294
	Pinaka [43]	✓							✓				✓					✓				272
	gazer-theta [1, 61]	✓	✓	✓			✓					✓	✓		✓	✓		✓			✓	255
	2ls [33, 76]	✓			✓	✓		✓					✓					✓				213
	UKojak [53, 82]		✓				✓						✓		✓	✓		✓				189
	Crux [51, 89]								✓									✓				176
	Korn [54]			✓			✓		✓									✓			✓	121
	Theta [90, 93]		✓	✓			✓					✓	✓			✓		✓		✓	✓	116
	Brick [34]	✓	✓			✓			✓									✓				99
	Graves-Par [8]																					93
	GDart-LLVM [9]								✓				✓									1



8 Dirk Beyer and Thomas Lemberger: Six Years Later: Testing vs. Model Checking

Table 3: Category results of the tools listed in Table 2. Only the best test generators (**T**) and model checkers (**M**) of each category are listed.

	Total Tasks	#Bugs Found	#Bugs Confirmed by Execution
Arrays			
T FuSeBMC	90	**90**	90
T FuSeBMC_IA	90	88	88
T VeriFuzz$^{Test-Comp}$	90	88	88
M VeriAbsL	90	81	76
M VeriAbs	90	80	66
M Bubaak	90	74	74
BitVectors			
T FuSeBMC	9	**9**	9
T FuSeBMC_IA	9	**9**	9
T VeriFuzz^{Both}	9	**9**	9
M Symbiotic$^{SV-COMP}$	9	8	8
M Esbmc-kind$^{SV-COMP}$	9	8	6
M Graves-CPA	9	8	6
ControlFlow			
T FuSeBMC	5	**5**	5
T FuSeBMC_IA	5	**5**	5
M Symbiotic^{Both}	5	**5**	5
M Bubaak	5	4	4
T VeriFuzz^{Both}	5	4	4
T Klee	5	4	4
M CVT-ParPort	5	4	3
ECA			
T VeriFuzz$^{SV-COMP}$	18	**15**	13
T Klee	18	14	14
M Bubaak	18	14	12
M Symbiotic$^{Test-Comp}$	18	13	13
M PeSCo	18	13	12
T FuSeBMC	18	12	12
Floats			
T FuSeBMC	32	**32**	32
T FuSeBMC_IA	32	31	31
T VeriFuzz$^{Test-Comp}$	32	31	31
M Brick	32	30	29
M CVT-ParPort	32	30	24
M CPAchecker	32	30	21
Hardware			
T VeriFuzz$^{Test-Comp}$	494	**319**	319
T FuSeBMC	494	288	288
T FuSeBMC_IA	494	288	288
M Graves-CPA	494	147	102
M CPAchecker	494	127	70
M PeSCo	494	109	61
Heap			
M Cbmc	47	**47**	43
M VeriAbs	47	47	33
M Bubaak	47	46	44
T FuSeBMC	47	45	45
T FuSeBMC_IA	47	45	45
T Klee	47	45	45
T VeriFuzz^{Both}	47	45	45
Loops			
T FuSeBMC	130	**128**	128
T FuSeBMC_IA	130	127	127
T VeriFuzz$^{Test-Comp}$	130	123	123
M VeriAbs	130	112	103
M VeriAbsL	130	100	86
M Korn	130	98	97
ProductLines			
T FuSeBMC	169	**169**	169
T FuSeBMC_IA	169	**169**	169
T Klee	169	**169**	169
T VeriFuzz^{Both}	169	**169**	169
M Bubaak	169	**169**	169
M VeriAbsL	169	**169**	169
M CVT-ParPort	169	169	167
Recursive			
T FuSeBMC	20	**19**	19
T FuSeBMC_IA	20	**19**	19
M Cbmc	20	**19**	19
M CVT-ParPort	20	**19**	19
M Graves-CPA	20	19	17
T VeriFuzz$^{Test-Comp}$	20	18	18
Sequentialized			
T VeriFuzz$^{Test-Comp}$	98	**95**	95
T FuSeBMC	98	94	94
T FuSeBMC_IA	98	92	92
M PeSCo	98	86	86
M CVT-ParPort	98	86	32
M Cbmc	98	85	29
XCSP			
M Cbmc	54	**50**	50
M CVT-AlgoSel	54	49	49
T VeriFuzz^{Both}	54	49	49
T WASP-C	54	49	49
M Esbmc-kind^{Both}	54	48	48
T FuSeBMC	54	47	47
BusyBox			
T FuSeBMC	5	**1**	1
T Klee	5	**1**	1
M PeSCo	5	1	0
Total			
T VeriFuzz$^{Test-Comp}$	1 173	**964**	964
T FuSeBMC	1 173	939	939
T FuSeBMC_IA	1 173	931	931
M PeSCo	1 173	667	475
M CPAchecker	1 173	665	458
M Esbmc-kind$^{SV-COMP}$	1 173	660	529

Table 4: Number of bugs found by the best tool of each category, the union of all test generators (**T**), the union of all model checkers (**M**), and all tools.

Category	Best Tool	All **T**	All **M**	All Tools
Arrays	90	87	90	90
BitVectors	9	9	9	9
ControlFlow	5	5	5	5
ECA	15	15	14	**17**
Floats	32	32	32	32
Hardware	319	340	175	**342**
Heap	47	45	47	47
Loops	128	128	127	128
ProductLines	169	169	169	169
Recursive	19	19	**20**	20
Sequentialized	95	95	90	95
XCSP	50	**51**	50	51
BusyBox	1	1	1	**2**

omit the category DeviceDriversLinux64 because no tool was able to find a bug in it.

The table shows that, for bug finding, individual test generators perform either better or as good as individual model checkers in all categories but Heap and XCSP. A clear divide between test generators and model checkers exists in four categories: In Arrays, the best test generator of that category, FuSeBMC, finds a bug in 90 tasks, while the best model checker of that category, VeriAbsL, finds a bug in only 81 tasks. In Hardware, VeriFuzz finds a bug in 319 tasks, while Graves-CPA finds a bug in only 147 tasks. In Loops, FuSeBMC finds a bug in 128 tasks while VeriAbs finds a bug in only 112 tasks. In Sequentialized, VeriFuzz finds a bug in 95 tasks while PeSCo finds a bugs in only 86 tasks.

The presented data answers our first research question with ‘yes’: At the current state-of-the-art for C, test generators perform significantly better in bug hunting than model checkers.

In our previous research study [26], the different tools complemented each other well, so that the combination of multiple tools yielded significant improvements in the number of bugs found. This is not true for the current results: Table 4 shows for each benchmark category the number of bugs found by the best tool in that category, the union of distinct bugs found by all test generators together (All **T**), the union of distinct bugs found by all model checkers together (All **M**), and the union of all considered tools (All Tools). The table shows that the unions only yield an improvement in 5 of the 13 categories, and that these improvements are also small. We explain this with the fact that, in contrast to the previous study, almost all currently considered tools already combine multiple approaches internally (cf. Table 2), rendering further external combinations effectless.

RQ 2. Can bug reports of software model checkers be validated through execution? Since a failing program execution provides the highest level of confidence in a verification result, we separately check how many of the confirmed verification results were confirmed not only by a third-party tool, but by actual program execution.

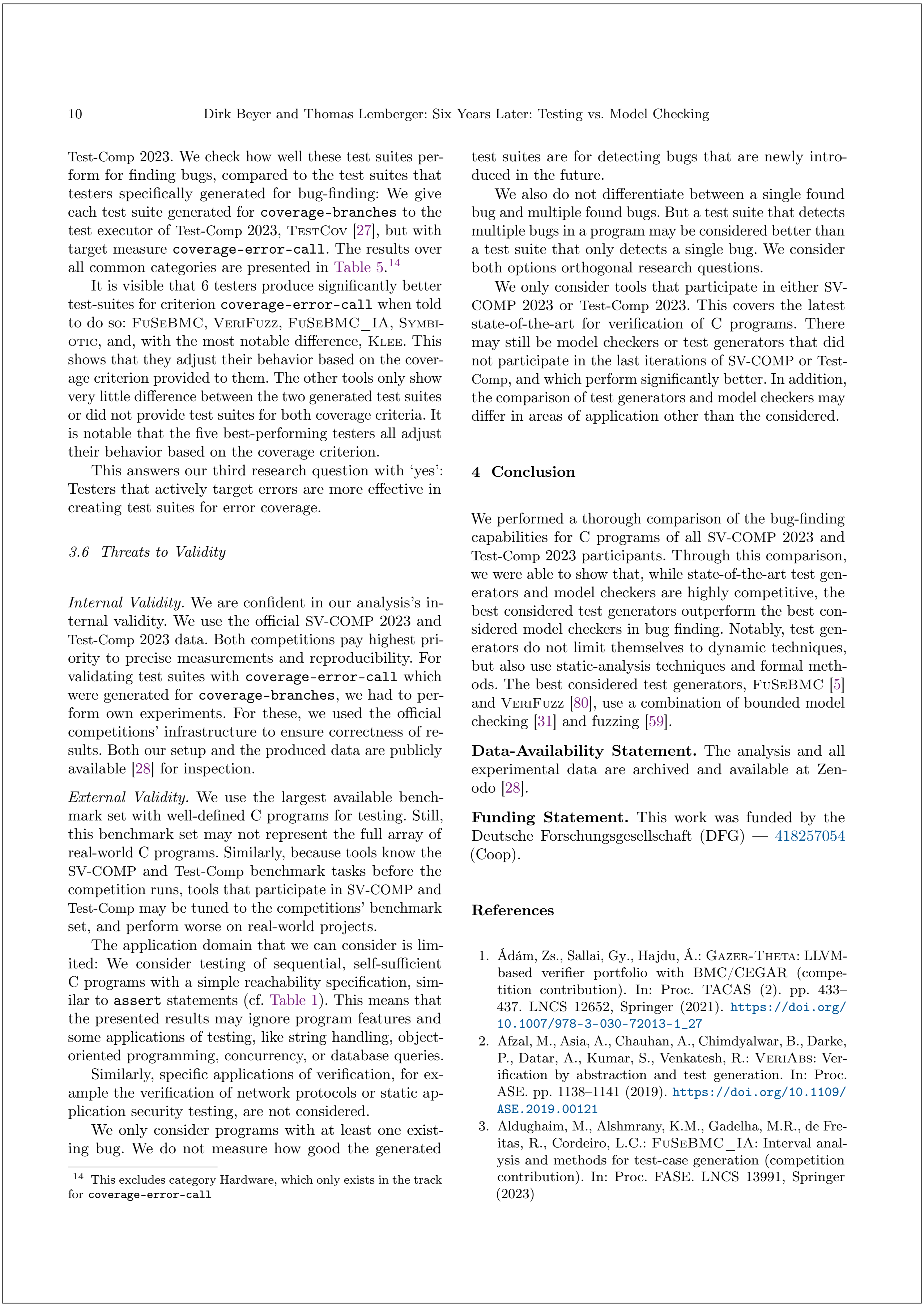
Table 5: Bug-finding capabilities of generated test suites that are targeted at either `coverage-error-call` or `coverage-branches`. The results exclude category Hardware because it is not part of the Test-Comp 2023 track on branch coverage.

Tools	Total Tasks	#Bugs Found `error-call`	#Bugs Found `branches`
FuSeBMC	679	**651**	594
VeriFuzz	679	**645**	611
FuSeBMC_IA	679	**643**	594
Klee	679	**541**	285
CoVeriTest	679	**479**	476
Symbiotic	679	**476**	456
TracerX	679	-	420
HybridTiger	679	**362**	281
WASP-C	679	354	**355**
Legion/SymCC	679	279	**281**
Esbmc-kind	679	352	-
PRTest	679	236	236
Legion	679	**108**	107

For this, we use the two execution-based witness validators of SV-COMP 2023, CPA-witness2test and FShell-witness2test. Table 3 shows in its last columns the number of found bugs that are confirmed through program execution. It is visible that the confirmation rate can be very high, for example for Brick in category Floats (29 of 30), for Cbmc in categories Heap (43 of 47), Recursive (19 of 19) and XCSP (50 of 50), or for PeSCo in category Sequentialized (86 of 86). On the other hand, the confirmation rate can also be very low, even for model checkers that perform well otherwise and in categories that other model checkers perform well in: Cbmc gets only 29 of 85 results confirmed through execution in category Sequentialized, and PeSCo gets only 61 of 109 results confirmed in category Hardware. This hints to bug reports (in the form of violation witnesses) that miss input values.

Thus, our answer to the second research question: The data shows that the execution-based validation of verification results is feasible and works well to provide a similar level of confidence in the result of model checkers as in test generators. But at the current state-of-the-art, model checkers have to produce more precise violation witnesses to offer the same level of confidence as test generators.

RQ 3. Are test generators that target errors more effective in finding bugs than test generators that target branch coverage? To answer our last research question, we consider the test suites [17] that each test generator generated for coverage criterion `coverage-branches` in



Test-Comp 2023. We check how well these test suites perform for finding bugs, compared to the test suites that testers specifically generated for bug-finding: We give each test suite generated for `coverage-branches` to the test executor of Test-Comp 2023, TestCov [27], but with target measure `coverage-error-call`. The results over all common categories are presented in Table 5.[14]

It is visible that 6 testers produce significantly better test-suites for criterion `coverage-error-call` when told to do so: FuSeBMC, VeriFuzz, FuSeBMC_IA, Symbiotic, and, with the most notable difference, Klee. This shows that they adjust their behavior based on the coverage criterion provided to them. The other tools only show very little difference between the two generated test suites or did not provide test suites for both coverage criteria. It is notable that the five best-performing testers all adjust their behavior based on the coverage criterion.

This answers our third research question with 'yes': Testers that actively target errors are more effective in creating test suites for error coverage.

3.6 Threats to Validity

Internal Validity. We are confident in our analysis's internal validity. We use the official SV-COMP 2023 and Test-Comp 2023 data. Both competitions pay highest priority to precise measurements and reproducibility. For validating test suites with `coverage-error-call` which were generated for `coverage-branches`, we had to perform own experiments. For these, we used the official competitions' infrastructure to ensure correctness of results. Both our setup and the produced data are publicly available [28] for inspection.

External Validity. We use the largest available benchmark set with well-defined C programs for testing. Still, this benchmark set may not represent the full array of real-world C programs. Similarly, because tools know the SV-COMP and Test-Comp benchmark tasks before the competition runs, tools that participate in SV-COMP and Test-Comp may be tuned to the competitions' benchmark set, and perform worse on real-world projects.

The application domain that we can consider is limited: We consider testing of sequential, self-sufficient C programs with a simple reachability specification, similar to `assert` statements (cf. Table 1). This means that the presented results may ignore program features and some applications of testing, like string handling, object-oriented programming, concurrency, or database queries.

Similarly, specific applications of verification, for example the verification of network protocols or static application security testing, are not considered.

We only consider programs with at least one existing bug. We do not measure how good the generated test suites are for detecting bugs that are newly introduced in the future.

We also do not differentiate between a single found bug and multiple found bugs. But a test suite that detects multiple bugs in a program may be considered better than a test suite that only detects a single bug. We consider both options orthogonal research questions.

We only consider tools that participate in either SV-COMP 2023 or Test-Comp 2023. This covers the latest state-of-the-art for verification of C programs. There may still be model checkers or test generators that did not participate in the last iterations of SV-COMP or Test-Comp, and which perform significantly better. In addition, the comparison of test generators and model checkers may differ in areas of application other than the considered.

4 Conclusion

We performed a thorough comparison of the bug-finding capabilities for C programs of all SV-COMP 2023 and Test-Comp 2023 participants. Through this comparison, we were able to show that, while state-of-the-art test generators and model checkers are highly competitive, the best considered test generators outperform the best considered model checkers in bug finding. Notably, test generators do not limit themselves to dynamic techniques, but also use static-analysis techniques and formal methods. The best considered test generators, FuSeBMC [5] and VeriFuzz [80], use a combination of bounded model checking [31] and fuzzing [59].

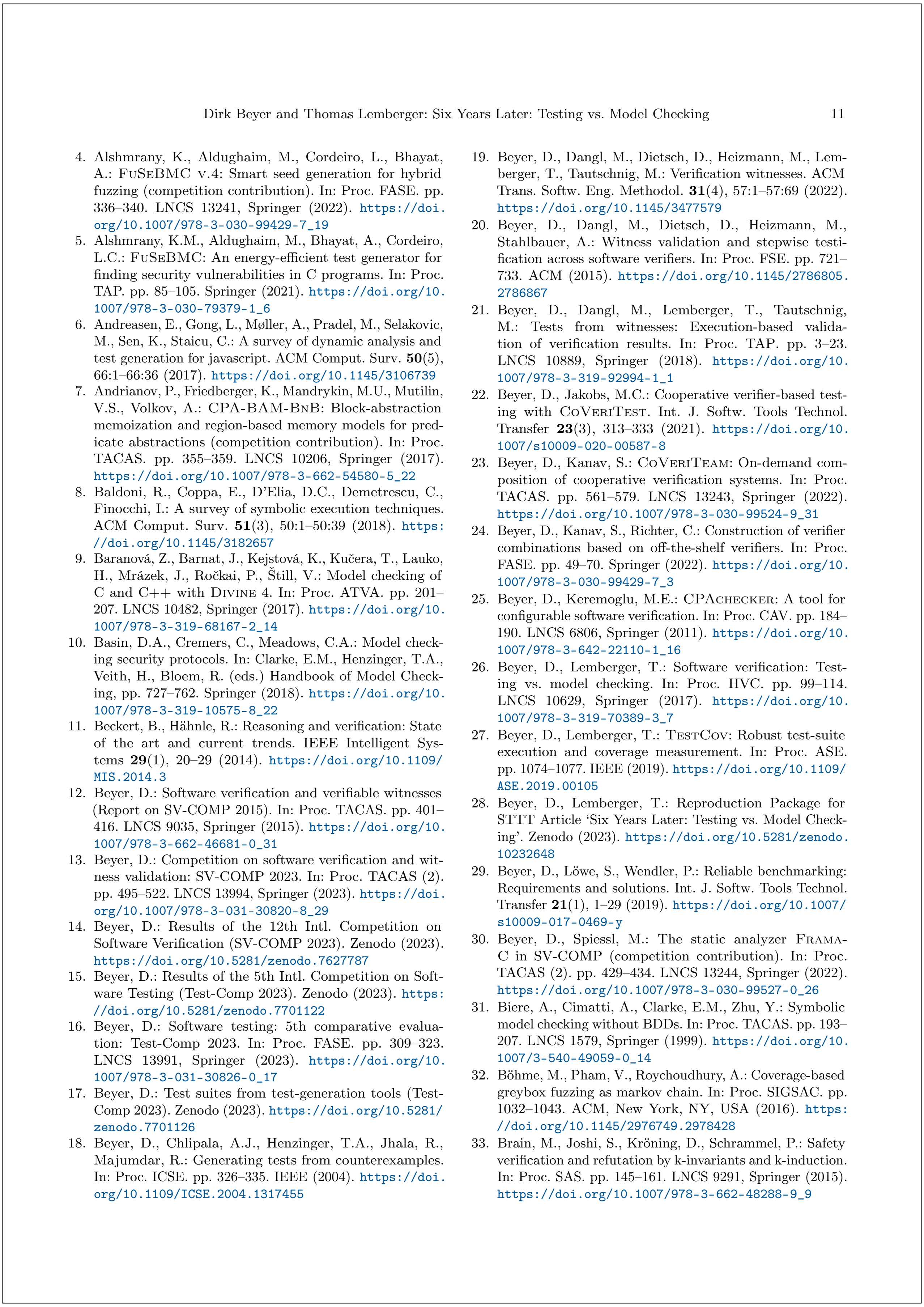
Data-Availability Statement. The analysis and all experimental data are archived and available at Zenodo [28].

Funding Statement. This work was funded by the Deutsche Forschungsgesellschaft (DFG) — 418257054 (Coop).

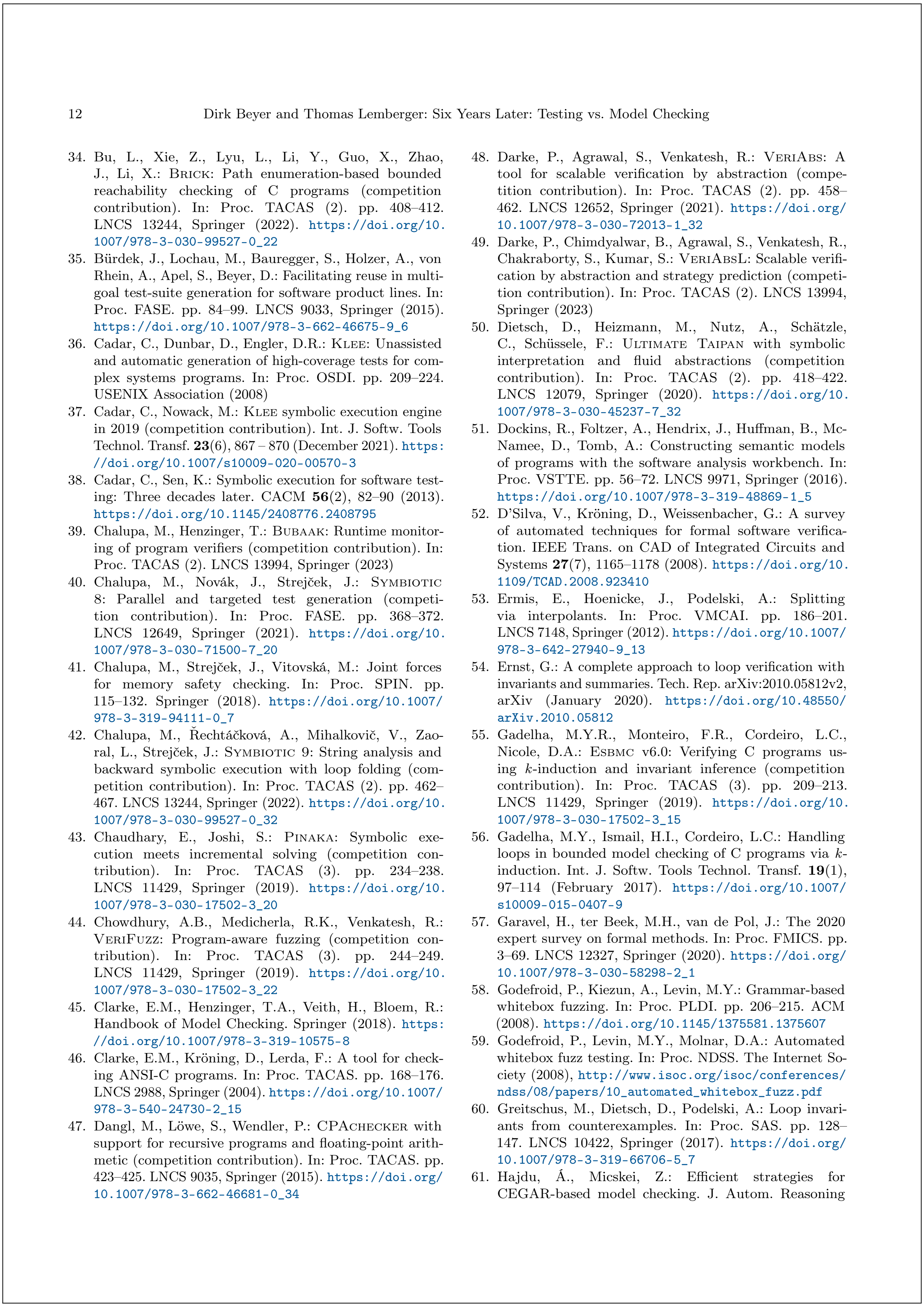
References

1. Ádám, Zs., Sallai, Gy., Hajdu, Á.: Gazer-Theta: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In: Proc. TACAS (2). pp. 433–437. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_27
2. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141 (2019). https://doi.org/10.1109/ASE.2019.00121
3. Alduhaim, M., Alshmrany, K.M., Gadelha, M.R., de Freitas, R., Cordeiro, L.C.: FuSeBMC_IA: Interval analysis and methods for test-case generation (competition contribution). In: Proc. FASE. LNCS 13991, Springer (2023)

[14] This excludes category Hardware, which only exists in the track for `coverage-error-call`



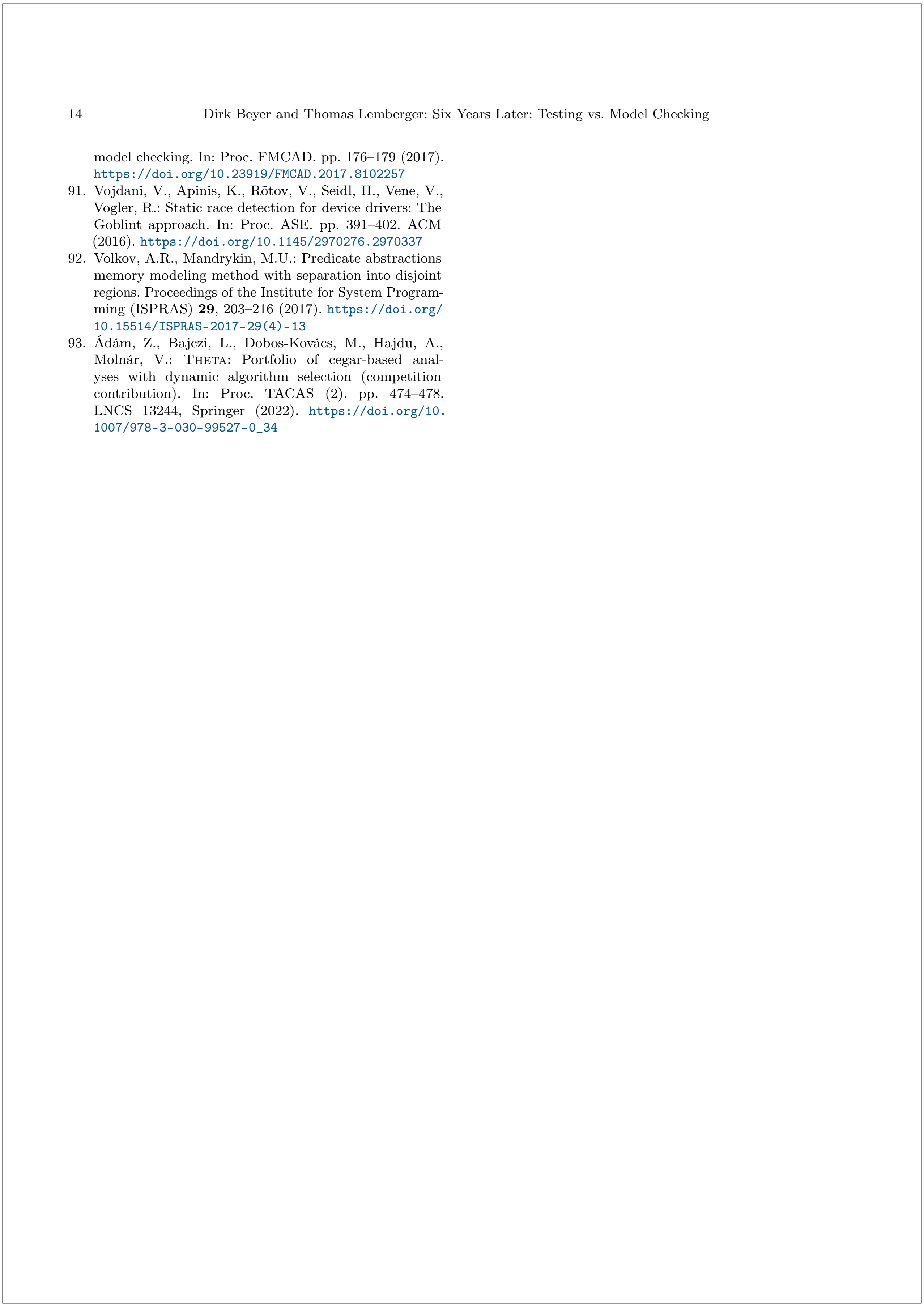
4. Alshmrany, K., Aldughaim, M., Cordeiro, L., Bhayat, A.: FuSeBMC v.4: Smart seed generation for hybrid fuzzing (competition contribution). In: Proc. FASE. pp. 336–340. LNCS 13241, Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_19
5. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Proc. TAP. pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6
6. Andreasen, E., Gong, L., Møller, A., Pradel, M., Selakovic, M., Sen, K., Staicu, C.: A survey of dynamic analysis and test generation for javascript. ACM Comput. Surv. **50**(5), 66:1–66:36 (2017). https://doi.org/10.1145/3106739
7. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS. pp. 355–359. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_22
8. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). https://doi.org/10.1145/3182657
9. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with Divine 4. In: Proc. ATVA. pp. 201–207. LNCS 10482, Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
10. Basin, D.A., Cremers, C., Meadows, C.A.: Model checking security protocols. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 727–762. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_22
11. Beckert, B., Hähnle, R.: Reasoning and verification: State of the art and current trends. IEEE Intelligent Systems **29**(1), 20–29 (2014). https://doi.org/10.1109/MIS.2014.3
12. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
13. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
14. Beyer, D.: Results of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627787
15. Beyer, D.: Results of the 5th Intl. Competition on Software Testing (Test-Comp 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7701122
16. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Proc. FASE. pp. 309–323. LNCS 13991, Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_17
17. Beyer, D.: Test suites from test-generation tools (Test-Comp 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7701126
18. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
19. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
20. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
21. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
22. Beyer, D., Jakobs, M.C.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transfer **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8
23. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
24. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Proc. FASE. pp. 49–70. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_3
25. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
26. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC. pp. 99–114. LNCS 10629, Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_7
27. Beyer, D., Lemberger, T.: TestCov: Robust test-suite execution and coverage measurement. In: Proc. ASE. pp. 1074–1077. IEEE (2019). https://doi.org/10.1109/ASE.2019.00105
28. Beyer, D., Lemberger, T.: Reproduction Package for STTT Article 'Six Years Later: Testing vs. Model Checking'. Zenodo (2023). https://doi.org/10.5281/zenodo.10232648
29. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
30. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 429–434. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_26
31. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
32. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Proc. SIGSAC. pp. 1032–1043. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978428
33. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9



34. Bu, L., Xie, Z., Lyu, L., Li, Y., Guo, X., Zhao, J., Li, X.: Brick: Path enumeration-based bounded reachability checking of C programs (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_22
35. Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D.: Facilitating reuse in multi-goal test-suite generation for software product lines. In: Proc. FASE. pp. 84–99. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_6
36. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
37. Cadar, C., Nowack, M.: Klee symbolic execution engine in 2019 (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 867 – 870 (December 2021). https://doi.org/10.1007/s10009-020-00570-3
38. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. CACM **56**(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
39. Chalupa, M., Henzinger, T.: Bubaak: Runtime monitoring of program verifiers (competition contribution). In: Proc. TACAS (2). LNCS 13994, Springer (2023)
40. Chalupa, M., Novák, J., Strejček, J.: Symbiotic 8: Parallel and targeted test generation (competition contribution). In: Proc. FASE. pp. 368–372. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_20
41. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
42. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (competition contribution). In: Proc. TACAS (2). pp. 462–467. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32
43. Chaudhary, E., Joshi, S.: Pinaka: Symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS (3). pp. 234–238. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_20
44. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program-aware fuzzing (competition contribution). In: Proc. TACAS (3). pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
45. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
46. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
47. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
48. Darke, P., Agrawal, S., Venkatesh, R.: VeriAbs: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32
49. Darke, P., Chimdyalwar, B., Agrawal, S., Venkatesh, R., Chakraborty, S., Kumar, S.: VeriAbsL: Scalable verification by abstraction and strategy prediction (competition contribution). In: Proc. TACAS (2). LNCS 13994, Springer (2023)
50. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate Taipan with symbolic interpretation and fluid abstractions (competition contribution). In: Proc. TACAS (2). pp. 418–422. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_32
51. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Proc. VSTTE. pp. 56–72. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_5
52. D'Silva, V., Kröning, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems **27**(7), 1165–1178 (2008). https://doi.org/10.1109/TCAD.2008.923410
53. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Proc. VMCAI. pp. 186–201. LNCS 7148, Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_13
54. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. arXiv:2010.05812v2, arXiv (January 2020). https://doi.org/10.48550/arXiv.2010.05812
55. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: Esbmc v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS (3). pp. 209–213. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15
56. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (February 2017). https://doi.org/10.1007/s10009-015-0407-9
57. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: Proc. FMICS. pp. 3–69. LNCS 12327, Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
58. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: Proc. PLDI. pp. 206–215. ACM (2008). https://doi.org/10.1145/1375581.1375607
59. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS. The Internet Society (2008), http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf
60. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS. pp. 128–147. LNCS 10422, Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
61. Hajdu, Á., Micskei, Z.: Efficient strategies for CEGAR-based model checking. J. Autom. Reasoning

64(6), 1051–1091 (2020). https://doi.org/10.1007/s10817-019-09535-x

62. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2). pp. 447–451. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
63. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
64. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Proc. VMCAI. pp. 151–166. LNCS 5403, Springer (2009). https://doi.org/10.1007/978-3-540-93900-9_15
65. Jaffar, J., Maghareh, R., Godboley, S., Ha, X.L.: TracerX: Dynamic symbolic execution with interpolation (competition contribution). In: Proc. FASE. pp. 530–534. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_28
66. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: a symbolic execution tool for verification. In: Proc. CAV. pp. 758–766. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_61
67. Jakobs, M.C., Richter, C.: CoVeriTest with adaptive time scheduling (competition contribution). In: Proc. FASE. pp. 358–362. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_18
68. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
69. Kettl, M., Lemberger, T.: The static analyzer Infer in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 451–456. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_30
70. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
71. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. LNCS 11187, Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_17
72. Leeson, W., Dwyer, M.: Graves-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
73. Lemberger, T.: Plain random test generation with PRTest (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 871–873 (December 2021). https://doi.org/10.1007/s10009-020-00568-x
74. Liu, D., Ernst, G., Murray, T., Rubinstein, B.: Legion: Best-first concolic testing (competition contribution). In: Proc. FASE. pp. 545–549. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_31
75. Liu, D., Ernst, G., Murray, T., Rubinstein, B.I.P.: Legion: Best-first concolic testing. In: Proc. ASE. pp. 54–65. IEEE (2020). https://doi.org/10.1145/3324884.3416629
76. Malík, V., Schrammel, P., Vojnar, T.: 2ls: Heap analysis and memory safety (competition contribution). In: Proc. TACAS (2). pp. 368–372. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_22
77. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Trans. Software Eng. **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563
78. McMinn, P.: Search-based software test-data generation: A survey. STVR **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
79. Metta, R., Medicherla, R.K., Chakraborty, S.: BMC+Fuzz: Efficient and effective test generation. In: Proc. DATE. pp. 1419–1424. IEEE (2022). https://doi.org/10.23919/DATE54114.2022.9774672
80. Metta, R., Medicherla, R.K., Karmarkar, H.: VeriFuzz: Fuzz centric test generation tool (competition contribution). In: Proc. FASE. pp. 341–346. LNCS 13241, Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_20
81. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-C: Modular domains and relational abstract interpretation for C programs (competition contribution). In: Proc. TACAS (2). LNCS 13994, Springer (2023)
82. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: Ultimate Kojak with memory safety checks (competition contribution). In: Proc. TACAS. pp. 458–460. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44
83. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six - mutation testing advances: An analysis and survey. Adv. Comput. **112**, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015
84. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4), 339–353 (2009). https://doi.org/10.1007/s10009-009-0118-1
85. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
86. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
87. Ruland, S., Lochau, M., Jakobs, M.C.: HybridTiger: Hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution). In: Proc. FASE. pp. 520–524. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_26
88. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints (competition contribution). In: Proc. TACAS (2). pp. 438–442. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28
89. Scott, R., Dockins, R., Ravitch, T., Tomb, A.: Crux: Symbolic execution meets SMT-based verification (competition contribution). Zenodo (February 2022). https://doi.org/10.5281/zenodo.6147218
90. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: A framework for abstraction refinement-based



model checking. In: Proc. FMCAD. pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257

91. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
92. Volkov, A.R., Mandrykin, M.U.: Predicate abstractions memory modeling method with separation into disjoint regions. Proceedings of the Institute for System Programming (ISPRAS) **29**, 203–216 (2017). https://doi.org/10.15514/ISPRAS-2017-29(4)-13
93. Ádám, Z., Bajczi, L., Dobos-Kovács, M., Hajdu, A., Molnár, V.: Theta: Portfolio of cegar-based analyses with dynamic algorithm selection (competition contribution). In: Proc. TACAS (2). pp. 474–478. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_34

2018 ACM/IEEE 40th International Conference on Software Engineering

Reducer-Based Construction of Conditional Verifiers

Dirk Beyer
LMU Munich, Germany

Marie-Christine Jakobs*
LMU Munich, Germany

Thomas Lemberger
LMU Munich, Germany

Heike Wehrheim*
Paderborn University, Germany

ABSTRACT

Despite recent advances, software verification remains challenging. To solve hard verification tasks, we need to leverage not just one but several different verifiers employing different technologies. To this end, we need to exchange information between verifiers. Conditional model checking was proposed as a solution to exactly this problem: The idea is to let the first verifier output a *condition* which describes the state space that it successfully verified and to instruct the second verifier to verify the yet unverified state space using this condition. However, most verifiers do not understand conditions as input.

In this paper, we propose the usage of an off-the-shelf construction of a *conditional verifier* from a given traditional verifier and a reducer. The reducer takes as input the program to be verified and the condition, and outputs a residual program whose paths cover the unverified state space described by the condition. As a proof of concept, we designed and implemented one particular reducer and composed three conditional model checkers from the three best verifiers at SV-COMP 2017. We defined a set of claims and experimentally evaluated their validity. All experimental data and results are available for replication.

CCS CONCEPTS

• **Software and its engineering** → **Formal methods**; **Formal software verification**;

KEYWORDS

Conditional Model Checking, Formal Verification, Testing, Program Analysis, Software Verification, Sequential Combination

ACM Reference Format:
Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. 2018. Reducer-Based Construction of Conditional Verifiers. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180259

1 INTRODUCTION

Software model checking [47] has received lots of attention in academia and industry [2, 48] in the past two decades — yet, there are many programs that are in principle verifiable, but no existing verifier can solve them automatically. There are many different approaches, but none is superior. The competition on software verification (SV-COMP) [5] gives a yearly overview over the state of the art, in terms of both strengths of verifiers on various categories and weaknesses as shown by a large amount of unsolved problems.

One promising idea is to combine the strengths of different verifiers by condition passing, which was formalized as *conditional model checking* (CMC) [10] six years ago. The idea is simple and effective: The first verifier reports what it had successfully verified and summarizes its work done as a *condition.* The next verifier reads the condition and verifies only the part of the state space not yet covered by the condition. This technique was shown to be effective, and sometimes even more efficient. Unfortunately, it is difficult to write a verifier that can parse the complicated conditions and effectively reduce the state space of the verifier. This complication is responsible for the situation that the technique is not as widely applied as it could be: only a few conditional model checkers exist.

To solve this problem, we developed an automatic construction template that can be used to construct a conditional verifier from a given arbitrary classical verifier. The original work proposed to run a product analysis that guides the state-space exploration such that it concentrates on the state space not covered by the condition. We propose an alternative solution, inspired by earlier work on conditional model checking and testing [35]: We define a *program reducer*, which takes as input a program and a condition, and computes a program whose executions are restricted to those not yet covered by the given condition. Having developed this component once, it is easy to construct a new conditional verifier using the equation CMC = $V \circ R$, where R is the reducer, V is an arbitrary verifier, and $\circ$ is the sequential application of first R to a given program and then V to the output program of R. The new verifier CMC is a conditional verifier that takes as input a program and a condition. Figure 1 illustrates this composition visually. There can be different implementations of reducers, and the reducers might leverage a notion of abstraction, causing the residual program to be more compact but less precise. We implemented a reducer that is based on a product construction, i.e., program and condition

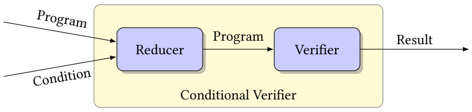


Figure 1: Construction of a conditional verifier

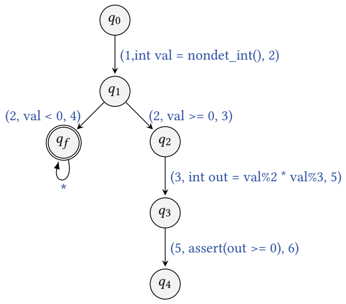
[*] This author was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE 2018, May 27 – June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180259

```
1  int val = nondet_int();
2  if (val >= 0) {
3    int out = val%2 * val%3;
   }
   else {
4    int out = -val;
   }
5  assert (out >= 0);
6
```

(a) Source code



(b) Condition

```
1  int val = nondet_int();
2  if (val >= 0) {
3     int out = val%2 * val%3;
4     assert (out >=0);
5   }
6   else
7      ;
```

(c) Residual program

Figure 2: Example: (a) fragment of a C program, (b) condition generated by CPAchecker with accepting states as double circles, assumptions elided (all *true*), label * subsuming all control-flow edges, and (c) residual program constructed by Reducer

are converted into automata and the reduced product automaton is converted back to a program.

In our study, we show that the construction works and is effective. We do not claim that our implementation of the reducer is the best possible, but we show for a number of verifiers how to increase the number of obtained results with the reducer-based construction of conditional verifiers. The approach can in some cases even reduce the resource consumption.

Contributions. We make the following contributions:

- We provide a reducer that understands more extensive conditions than a reducer that was previously used in the context of conditional model checking and testing [35].
- We construct a number of conditional verifiers from existing verifiers in order to experimentally show that new combinations with condition passing can significantly increase the number of verified programs.
- We apply the concept also to test-case generation and show that the construction effectively works.
- Our reducer and all experimental data are available for other researchers and practitioners for replication or to strengthen their own verification infrastructure by using newly constructed conditional verifiers that were not available before.

2 CONDITION-BASED REDUCERS

The objective of our work is the construction of conditional verifiers. Conditional verifiers are verification tools accepting programs together with conditions as input. A conditional verifier should check the parts of the program not covered by the condition. To this end, we employ *reducers* constructing residual programs from conditions. We start with giving a formal account of conditions and reducers. In our notation, we follow previous work [11].

2.1 Foundations

Programs are represented by *control-flow automata*[1] (CFAs) $C = (L, \ell_0, G)$ that consist of a set of locations L, an initial location ℓ_0, and a set of control-flow edges $G \subseteq L \times Ops \times L$, where Ops is the set of operations. Intuitively, a program and its CFA are semantically equivalent because the CFA contains exactly the operations of the program on its control-flow edges and in exactly the same order. Our construction of reducers relies on soundly converting programs to CFAs and back within tools. We let $\mathfrak{C}$ be the set of all CFAs. In our presentation, we consider operations from a simple programming language, with assume operations and assignments on integer variables. Our implementation covers C programs.

We let X be the set of variables occurring in the operations Ops. A concrete data state c is thus a mapping of X to $\mathbb{Z}$. A *concrete program path* of a CFA $C = (L, \ell_0, G)$ is a sequence $(c_0, \ell_0) \xrightarrow{g_1} \ldots \xrightarrow{g_n} (c_n, \ell_n)$ such that c_0 assigns 0 to all variables, $g_i = (\ell_{i-1}, op_i, \ell_i) \in G$, and $c_{i-1} \xrightarrow{op_i} c_i$, i.e., (a) in case of assume operations, $c_{i-1} \models op_i$ (op_i is the assumption) and $c_{i-1} = c_i$, and (b) in case of assignments, $c_i = \mathrm{SP}_{op_i}(c_{i-1})$, where SP is the strongest-post operator of the operational semantics. From a concrete program path $\pi = (c_0, \ell_0) \xrightarrow{g_1} \ldots \xrightarrow{g_n} (c_n, \ell_n)$, we can derive an *execution* $ex(\pi) = c_0 c_1 \ldots c_n$. We let $path(C)$ be the set of all concrete program paths and $ex(C)$ be the set of executions of a CFA C. A CFA C is *deterministic* (and hence representable as a C program) if the following holds for all $\ell \in L$, $(\ell, op_1, \ell_1), (\ell, op_2, \ell_2) \in G$: either $op_1 = op_2$ and $\ell_1 = \ell_2$, or op_1 is an assume operation and $op_1 \wedge op_2$ is unsatisfiable.

Conditions subsume the results of verification runs on programs. A condition basically states which paths have been explored. In addition, a condition might involve *assumptions* under which the verifier has explored a certain path. Assumptions are given as state conditions (from a set Φ). We write $c \models \varphi$ to say that a concrete state c satisfies a state condition φ.

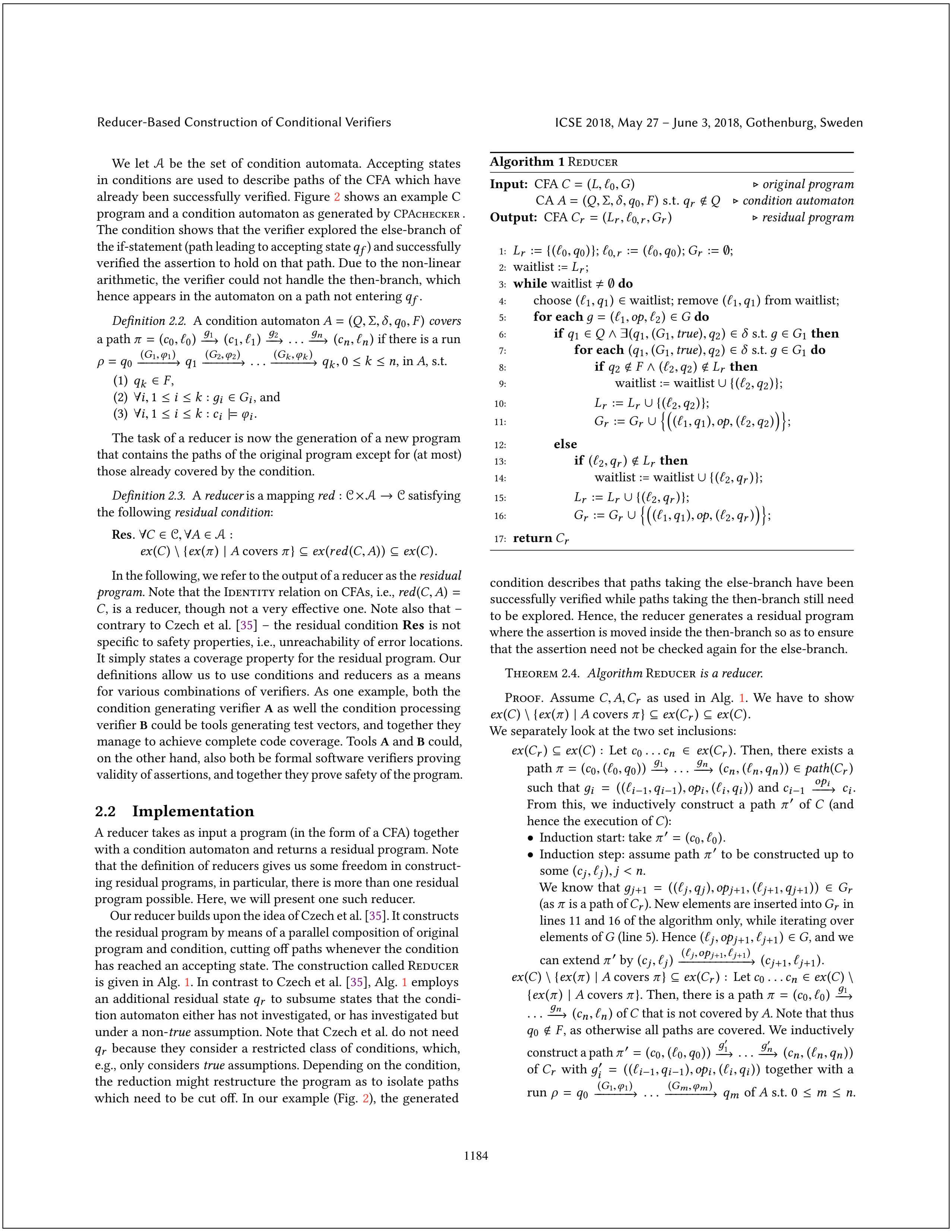
Definition 2.1. A *condition automaton* (CA) (short: condition) $A = (Q, \Sigma, \delta, q_0, F)$ consists of

- a finite set Q of *states* and an initial state $q_0 \in Q$,
- an *alphabet* $\Sigma \subseteq 2^G \times \Phi$,
- a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, and
- a set $F \subseteq Q$ of *accepting states*,

and satisfies the following well-formedness condition:

$$\neg \exists (q_f, *, q) \in \delta \text{ with } q_f \in F \wedge q \notin F\ .$$

[1] CFAs are a variant of control-flow graphs [1], with operations attached to the edges.



We let $\mathcal{A}$ be the set of condition automata. Accepting states in conditions are used to describe paths of the CFA which have already been successfully verified. Figure 2 shows an example C program and a condition automaton as generated by CPAchecker . The condition shows that the verifier explored the else-branch of the if-statement (path leading to accepting state q_f) and successfully verified the assertion to hold on that path. Due to the non-linear arithmetic, the verifier could not handle the then-branch, which hence appears in the automaton on a path not entering q_f.

Definition 2.2. A condition automaton $A = (Q, \Sigma, \delta, q_0, F)$ *covers* a path $\pi = (c_0, \ell_0) \xrightarrow{g_1} (c_1, \ell_1) \xrightarrow{g_2} \dots \xrightarrow{g_n} (c_n, \ell_n)$ if there is a run $\rho = q_0 \xrightarrow{(G_1, \varphi_1)} q_1 \xrightarrow{(G_2, \varphi_2)} \dots \xrightarrow{(G_k, \varphi_k)} q_k, 0 \leq k \leq n$, in A, s.t.

(1) $q_k \in F$,
(2) $\forall i, 1 \leq i \leq k : g_i \in G_i$, and
(3) $\forall i, 1 \leq i \leq k : c_i \models \varphi_i$.

The task of a reducer is now the generation of a new program that contains the paths of the original program except for (at most) those already covered by the condition.

Definition 2.3. A *reducer* is a mapping $red : \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{C}$ satisfying the following *residual condition*:

Res. $\forall C \in \mathcal{C}, \forall A \in \mathcal{A}$:
$ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\} \subseteq ex(red(C, A)) \subseteq ex(C)$.

In the following, we refer to the output of a reducer as the *residual program.* Note that the Identity relation on CFAs, i.e., $red(C, A) = C$, is a reducer, though not a very effective one. Note also that – contrary to Czech et al. [35] – the residual condition **Res** is not specific to safety properties, i.e., unreachability of error locations. It simply states a coverage property for the residual program. Our definitions allow us to use conditions and reducers as a means for various combinations of verifiers. As one example, both the condition generating verifier **A** as well the condition processing verifier **B** could be tools generating test vectors, and together they manage to achieve complete code coverage. Tools **A** and **B** could, on the other hand, also both be formal software verifiers proving validity of assertions, and together they prove safety of the program.

2.2 Implementation

A reducer takes as input a program (in the form of a CFA) together with a condition automaton and returns a residual program. Note that the definition of reducers gives us some freedom in constructing residual programs, in particular, there is more than one residual program possible. Here, we will present one such reducer.

Our reducer builds upon the idea of Czech et al. [35]. It constructs the residual program by means of a parallel composition of original program and condition, cutting off paths whenever the condition has reached an accepting state. The construction called Reducer is given in Alg. 1. In contrast to Czech et al. [35], Alg. 1 employs an additional residual state q_r to subsume states that the condition automaton either has not investigated, or has investigated but under a non-*true* assumption. Note that Czech et al. do not need q_r because they consider a restricted class of conditions, which, e.g., only considers *true* assumptions. Depending on the condition, the reduction might restructure the program as to isolate paths which need to be cut off. In our example (Fig. 2), the generated condition describes that paths taking the else-branch have been successfully verified while paths taking the then-branch still need to be explored. Hence, the reducer generates a residual program where the assertion is moved inside the then-branch so as to ensure that the assertion need not be checked again for the else-branch.

Algorithm 1 Reducer

```
Input:  CFA $C = (L, \ell_0, G)$                                  ▹ original program
        CA $A = (Q, \Sigma, \delta, q_0, F)$ s.t. $q_r \notin Q$   ▹ condition automaton
Output: CFA $C_r = (L_r, \ell_{0,r}, G_r)$                          ▹ residual program

1:  $L_r := \{(\ell_0, q_0)\}$; $\ell_{0,r} := (\ell_0, q_0)$; $G_r := \emptyset$;
2:  waitlist := $L_r$;
3:  while waitlist $\neq \emptyset$ do
4:      choose $(\ell_1, q_1) \in$ waitlist; remove $(\ell_1, q_1)$ from waitlist;
5:      for each $g = (\ell_1, op, \ell_2) \in G$ do
6:          if $q_1 \in Q \wedge \exists (q_1, (G_1, true), q_2) \in \delta$ s.t. $g \in G_1$ then
7:              for each $(q_1, (G_1, true), q_2) \in \delta$ s.t. $g \in G_1$ do
8:                  if $q_2 \notin F \wedge (\ell_2, q_2) \notin L_r$ then
9:                      waitlist := waitlist $\cup \{(\ell_2, q_2)\}$;
10:                 $L_r := L_r \cup \{(\ell_2, q_2)\}$;
11:                 $G_r := G_r \cup \Big\{\Big((\ell_1, q_1), op, (\ell_2, q_2)\Big)\Big\}$;
12:         else
13:             if $(\ell_2, q_r) \notin L_r$ then
14:                 waitlist := waitlist $\cup \{(\ell_2, q_r)\}$;
15:             $L_r := L_r \cup \{(\ell_2, q_r)\}$;
16:             $G_r := G_r \cup \Big\{\Big((\ell_1, q_1), op, (\ell_2, q_r)\Big)\Big\}$;
17: return $C_r$
```

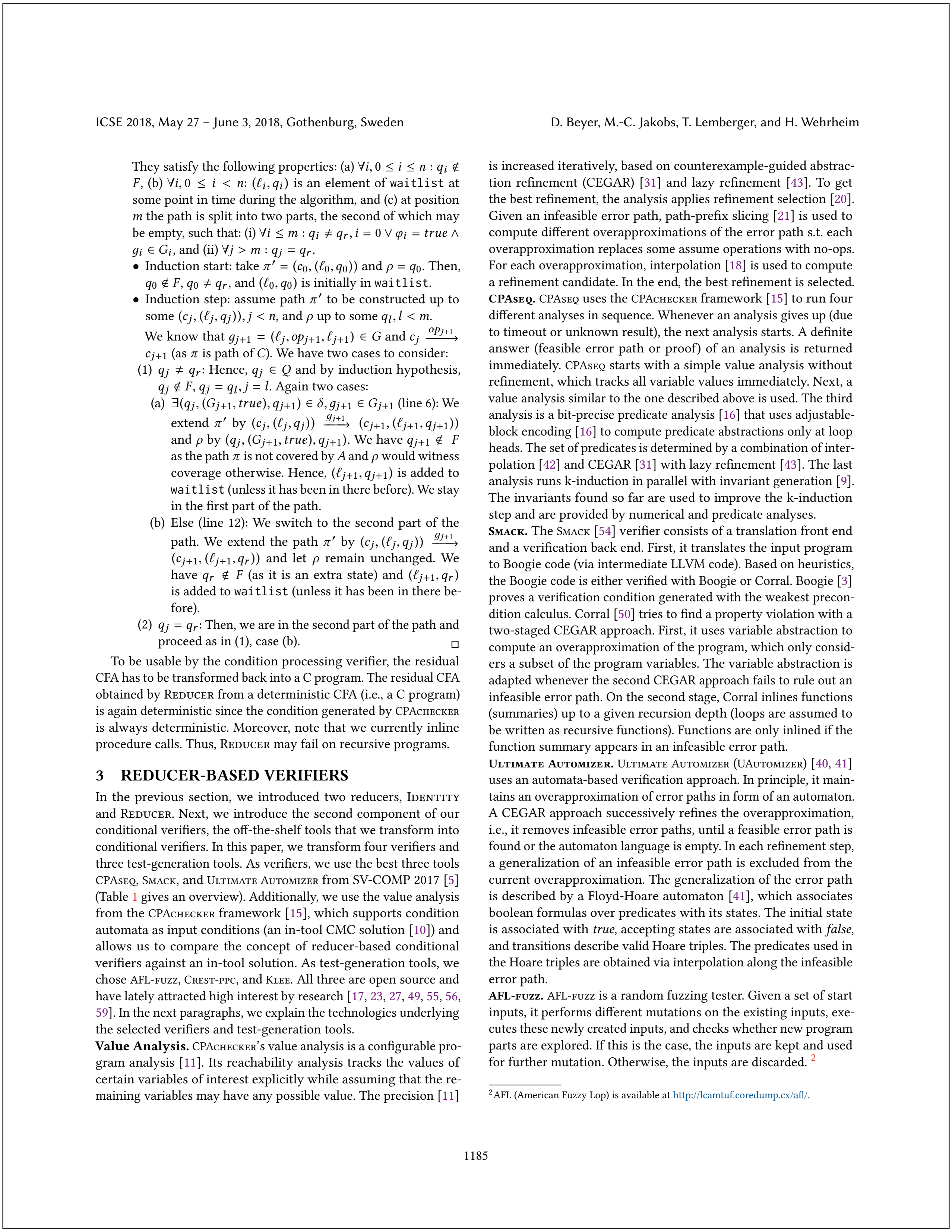
Theorem 2.4. *Algorithm* Reducer *is a reducer.*

Proof. Assume C, A, C_r as used in Alg. 1. We have to show $ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\} \subseteq ex(C_r) \subseteq ex(C)$.
We separately look at the two set inclusions:

$ex(C_r) \subseteq ex(C)$: Let $c_0 \dots c_n \in ex(C_r)$. Then, there exists a path $\pi = (c_0, (\ell_0, q_0)) \xrightarrow{g_1} \dots \xrightarrow{g_n} (c_n, (\ell_n, q_n)) \in path(C_r)$ such that $g_i = ((\ell_{i-1}, q_{i-1}), op_i, (\ell_i, q_i))$ and $c_{i-1} \xrightarrow{op_i} c_i$. From this, we inductively construct a path π' of C (and hence the execution of C):

- Induction start: take $\pi' = (c_0, \ell_0)$.
- Induction step: assume path π' to be constructed up to some $(c_j, \ell_j), j < n$.
  We know that $g_{j+1} = ((\ell_j, q_j), op_{j+1}, (\ell_{j+1}, q_{j+1})) \in G_r$ (as π is a path of C_r). New elements are inserted into G_r in lines 11 and 16 of the algorithm only, while iterating over elements of G (line 5). Hence $(\ell_j, op_{j+1}, \ell_{j+1}) \in G$, and we can extend π' by $(c_j, \ell_j) \xrightarrow{(\ell_j, op_{j+1}, \ell_{j+1})} (c_{j+1}, \ell_{j+1})$.

$ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\} \subseteq ex(C_r)$: Let $c_0 \dots c_n \in ex(C) \setminus \{ex(\pi) \mid A \text{ covers } \pi\}$. Then, there is a path $\pi = (c_0, \ell_0) \xrightarrow{g_1} \dots \xrightarrow{g_n} (c_n, \ell_n)$ of C that is not covered by A. Note that thus $q_0 \notin F$, as otherwise all paths are covered. We inductively construct a path $\pi' = (c_0, (\ell_0, q_0)) \xrightarrow{g'_1} \dots \xrightarrow{g'_n} (c_n, (\ell_n, q_n))$ of C_r with $g'_i = ((\ell_{i-1}, q_{i-1}), op_i, (\ell_i, q_i))$ together with a run $\rho = q_0 \xrightarrow{(G_1, \varphi_1)} \dots \xrightarrow{(G_m, \varphi_m)} q_m$ of A s.t. $0 \leq m \leq n$.



They satisfy the following properties: (a) $\forall i, 0 \leq i \leq n : q_i \notin F$, (b) $\forall i, 0 \leq i < n$: (ℓ_i, q_i) is an element of `waitlist` at some point in time during the algorithm, and (c) at position m the path is split into two parts, the second of which may be empty, such that: (i) $\forall i \leq m : q_i \neq q_r, i = 0 \vee \varphi_i = true \wedge g_i \in G_i$, and (ii) $\forall j > m : q_j = q_r$.

- Induction start: take $\pi' = (c_0, (\ell_0, q_0))$ and $\rho = q_0$. Then, $q_0 \notin F$, $q_0 \neq q_r$, and (ℓ_0, q_0) is initially in `waitlist`.
- Induction step: assume path π' to be constructed up to some $(c_j, (\ell_j, q_j)), j < n$, and ρ up to some $q_l, l < m$. We know that $g_{j+1} = (\ell_j, op_{j+1}, \ell_{j+1}) \in G$ and $c_j \xrightarrow{op_{j+1}} c_{j+1}$ (as π is path of C). We have two cases to consider:
  (1) $q_j \neq q_r$: Hence, $q_j \in Q$ and by induction hypothesis, $q_j \notin F, q_j = q_l, j = l$. Again two cases:
    (a) $\exists(q_j, (G_{j+1}, true), q_{j+1}) \in \delta, g_{j+1} \in G_{j+1}$ (line 6): We extend π' by $(c_j, (\ell_j, q_j)) \xrightarrow{g_{j+1}} (c_{j+1}, (\ell_{j+1}, q_{j+1}))$ and ρ by $(q_j, (G_{j+1}, true), q_{j+1})$. We have $q_{j+1} \notin F$ as the path π is not covered by A and ρ would witness coverage otherwise. Hence, (ℓ_{j+1}, q_{j+1}) is added to `waitlist` (unless it has been in there before). We stay in the first part of the path.
    (b) Else (line 12): We switch to the second part of the path. We extend the path π' by $(c_j, (\ell_j, q_j)) \xrightarrow{g_{j+1}} (c_{j+1}, (\ell_{j+1}, q_r))$ and let ρ remain unchanged. We have $q_r \notin F$ (as it is an extra state) and (ℓ_{j+1}, q_r) is added to `waitlist` (unless it has been in there before).
  (2) $q_j = q_r$: Then, we are in the second part of the path and proceed as in (1), case (b). □

To be usable by the condition processing verifier, the residual CFA has to be transformed back into a C program. The residual CFA obtained by Reducer from a deterministic CFA (i.e., a C program) is again deterministic since the condition generated by CPAchecker is always deterministic. Moreover, note that we currently inline procedure calls. Thus, Reducer may fail on recursive programs.

3 REDUCER-BASED VERIFIERS

In the previous section, we introduced two reducers, Identity and Reducer. Next, we introduce the second component of our conditional verifiers, the off-the-shelf tools that we transform into conditional verifiers. In this paper, we transform four verifiers and three test-generation tools. As verifiers, we use the best three tools CPAseq, Smack, and Ultimate Automizer from SV-COMP 2017 [5] (Table 1 gives an overview). Additionally, we use the value analysis from the CPAchecker framework [15], which supports condition automata as input conditions (an in-tool CMC solution [10]) and allows us to compare the concept of reducer-based conditional verifiers against an in-tool solution. As test-generation tools, we chose AFL-fuzz, Crest-ppc, and Klee. All three are open source and have lately attracted high interest by research [17, 23, 27, 49, 55, 56, 59]. In the next paragraphs, we explain the technologies underlying the selected verifiers and test-generation tools.

Value Analysis. CPAchecker's value analysis is a configurable program analysis [11]. Its reachability analysis tracks the values of certain variables of interest explicitly while assuming that the remaining variables may have any possible value. The precision [11] is increased iteratively, based on counterexample-guided abstraction refinement (CEGAR) [31] and lazy refinement [43]. To get the best refinement, the analysis applies refinement selection [20]. Given an infeasible error path, path-prefix slicing [21] is used to compute different overapproximations of the error path s.t. each overapproximation replaces some assume operations with no-ops. For each overapproximation, interpolation [18] is used to compute a refinement candidate. In the end, the best refinement is selected.

CPAseq. CPAseq uses the CPAchecker framework [15] to run four different analyses in sequence. Whenever an analysis gives up (due to timeout or unknown result), the next analysis starts. A definite answer (feasible error path or proof) of an analysis is returned immediately. CPAseq starts with a simple value analysis without refinement, which tracks all variable values immediately. Next, a value analysis similar to the one described above is used. The third analysis is a bit-precise predicate analysis [16] that uses adjustable-block encoding [16] to compute predicate abstractions only at loop heads. The set of predicates is determined by a combination of interpolation [42] and CEGAR [31] with lazy refinement [43]. The last analysis runs k-induction in parallel with invariant generation [9]. The invariants found so far are used to improve the k-induction step and are provided by numerical and predicate analyses.

Smack. The Smack [54] verifier consists of a translation front end and a verification back end. First, it translates the input program to Boogie code (via intermediate LLVM code). Based on heuristics, the Boogie code is either verified with Boogie or Corral. Boogie [3] proves a verification condition generated with the weakest precondition calculus. Corral [50] tries to find a property violation with a two-staged CEGAR approach. First, it uses variable abstraction to compute an overapproximation of the program, which only considers a subset of the program variables. The variable abstraction is adapted whenever the second CEGAR approach fails to rule out an infeasible error path. On the second stage, Corral inlines functions (summaries) up to a given recursion depth (loops are assumed to be written as recursive functions). Functions are only inlined if the function summary appears in an infeasible error path.

Ultimate Automizer. Ultimate Automizer (UAutomizer) [40, 41] uses an automata-based verification approach. In principle, it maintains an overapproximation of error paths in form of an automaton. A CEGAR approach successively refines the overapproximation, i.e., it removes infeasible error paths, until a feasible error path is found or the automaton language is empty. In each refinement step, a generalization of an infeasible error path is excluded from the current overapproximation. The generalization of the error path is described by a Floyd-Hoare automaton [41], which associates boolean formulas over predicates with its states. The initial state is associated with *true*, accepting states are associated with *false*, and transitions describe valid Hoare triples. The predicates used in the Hoare triples are obtained via interpolation along the infeasible error path.

AFL-fuzz. AFL-fuzz is a random fuzzing tester. Given a set of start inputs, it performs different mutations on the existing inputs, executes these newly created inputs, and checks whether new program parts are explored. If this is the case, the inputs are kept and used for further mutation. Otherwise, the inputs are discarded. [2]

[2] AFL (American Fuzzy Lop) is available at http://lcamtuf.coredump.cx/afl/.

Table 1: Overview of applied verification technologies in the verifiers

		Refinement			
Verifier	Technique	CEGAR	Lazy abstraction	Interpolation	Bitprecise
CPAseq	ARG, explicit and numerical values, predicates, k-induction	✓	✓	✓	✓
Smack	property-driven reachability [24], bounded model checking [22]	✓	✓	×	✓
UAutomizer	automata, predicates	✓	✓	✓	✓

Klee. Klee [26] uses symbolic execution for test-case generation. Symbolic execution is an extension to concrete execution of a program. For every unknown input value to a program, a new symbolic value is introduced that initially represents any possible value. During execution, the symbolic values are constrained by branching conditions along the program (e.g., if-branches in a C program). These constraints are used to compute whether a given program path is feasible, and which class of input values will lead to executions that take this path. Whenever both branches are feasible in a symbolic execution, Klee copies its current symbolic execution state and continues to explore one branch with the current state and the other with the copied state. After each step in a program, Klee heuristically chooses with which of the existing execution states to continue. Given several heuristics, Klee alternates between them.

Crest-ppc. Crest-ppc [49] is an improved version of Crest [25]. Crest uses concolic execution for testing and provides different heuristics to achieve higher code coverage. Concolic execution is a combination of symbolic execution and concrete execution. A program under test is executed with concrete inputs that determine one concrete execution path. In parallel, a symbolic execution is performed on that path to obtain constraints over program inputs on this path. Based on these *path constraints*, a constraint solver computes new inputs that lead to the execution of another, yet unvisited program part. New executions are performed and new inputs are generated until all program parts are explored. Crest uses heuristics to choose which unvisited program part to explore next. To increase the performance, Crest-ppc adds a heuristic to Crest that submits more calls to the constraint solver but uses fewer constraints per call.

4 EVALUATION

4.1 Claims to be Evaluated

In the following, we list our claims and how we plan to evaluate them. The claims are not on efficiency, but on effectiveness. That is, we provide means for solving additional verification tasks by investing more computing resources, but without implementing or changing verification tools.

Feasibility Hypothesis. A reducer can be used to effectively construct conditional verifiers from existing verification tools. *Evaluation Plan:* We show this by implementing one particular instance of a reducer, and apply our reducer-based construction of conditional verifiers to three model checkers and three testers. The result is a set of six conditional verifiers, and we take standard configurations "out of the box", without changing a single line of the verifiers.

Null Hypothesis. Applying a reducer has no effect. *Evaluation Plan:* We compare the results using our reducer against the results using the identity function as replacement for the reducer.

Claim 1. Reducer-based conditional verification is not much worse then "native" conditional verification. *Evaluation Plan:* The original proposal of CMC [10] implements the restriction of the state space that the condition describes internally in the exploration engine of the verifier. We claim that it also works reasonably well to use an external reducer instead, which opens the door for constructing new conditional verifiers without actual implementation work.

Claim 2. The technique of conditional verification can effectively increase the number of overall solved verification tasks if additional resources are provided. *Evaluation Plan:* We select a number of hard-to-solve verification tasks and perform experiments on them using the original verifiers and the constructed verifiers.

Claim 3. Conditional verification with condition passing can solve verification tasks that neither CPAseq, Smack, nor Ultimate Automizer can solve. *Evaluation Plan:* We select from a given set of verification tasks those verification tasks that none of the original verifiers, but at least one of the conditional verifiers can solve.

Claim 4. The use of different conditional verifiers improves the overall effectiveness. *Evaluation Plan:* We report results for different conditional verifiers and consider verification tasks that only one conditional combination can solve.

Claim 5. Reducer-based conditional verification is also applicable to test-case generation. *Evaluation Plan:* We construct conditional verifiers from three test-generation tools and compare the number of generated crashing tests against the result of the test-generation tools alone.

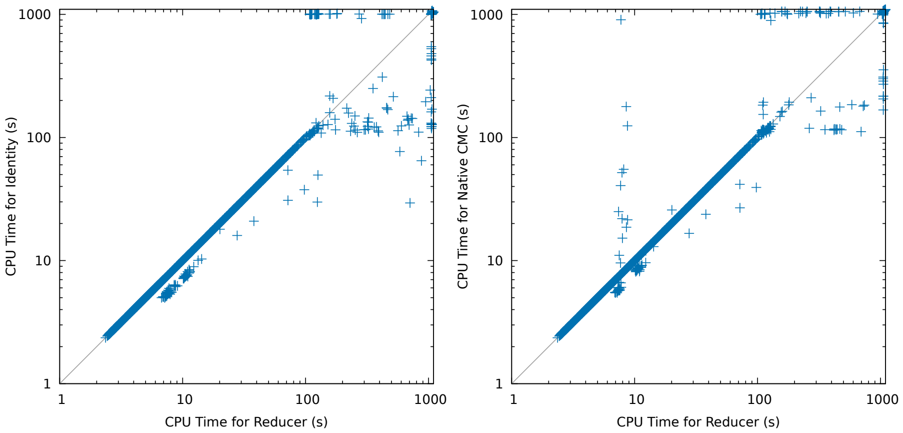
4.2 Setup

Computing Resources. We performed our experiments on machines with an Intel Xeon E3-1230 v5 CPU with 8 processing units each, a frequency of 3.4 GHz, 33 GB of memory, and an Ubuntu 16.04 operating system with Linux kernel 4.4. We limited each analysis run to 15 GB of memory and a varying time limit, depending on the experiment, and allowed it to use all 8 processing units. We report CPU time and memory use with two significant digits.

Verification Tasks. To get a representative set of verification tasks, we used all 5 687 programs from ReachSafety categories of the SV-COMP benchmark set[3] in revision cc49668 [4]. For all input programs, we verify the property that function __VERIFIER_error is never called. A total of 1 501 of the 5 687 programs are unsafe, i.e., the call to __VERIFIER_error is reachable, and 4 186 programs are safe.

Tools. We used a predicate analysis for condition generation and a value analysis for comparison with native conditional model checking, both from the CPAchecker project. Our implementation of a reducer is also available in the CPAchecker project. For all experiments, we used CPAchecker from branch reducer-patch in revision r25656. For the verifiers in the composition of our reducer with a verifier, we use the three best tools from SV-COMP 2017, as submitted to the competition[5] (without any modifications) and

[3] https://sv-comp.sosy-lab.org/2017/benchmarks.php
[4] https://github.com/sosy-lab/sv-benchmarks
[5] https://sv-comp.sosy-lab.org/2017/systems.php



(a) Reducer vs. Identity (pure sequential combination) (b) Reducer vs. native CMC implementation in CPAchecker

Figure 3: Comparison of CPU time of different CMC solutions for predicate (100s) + value analysis

the three test-generation tools described previously. To streamline the testing process for the test-generation tools, we use the testing framework tbf [17] [6] in revision b60a924. We run our experiments with BenchExec [19] (version 1.14).[7]

Availability. All our experimental data are available online [14].[8]

4.3 Experiments

Feasibility Hypothesis. We designed and implemented a proof-of concept reducer, and licensed the reducer using the open-source license Apache 2.0 such that other researchers can later use it. While our implementation certainly has potential for improvement, we show that the approach of composing a conditional verifier from an arbitrary verifier and our reducer works in practice. We demonstrated this by using the three best verifiers directly from the SV-COMP web site and composed the conditional verifiers without any change to the verifiers. In addition, we also composed conditional verifiers from test-generation tools, in order to help test-generators to produce crashing tests for more verification tasks.

Null Hypothesis. We have experimented with verification runs in which we replaced our reducer by an identity function Identity, i.e., the reducer is effectively removed from the tool chain. The first verifier, which generates the condition, is a predicate analysis that we restrict to at most 100 s of CPU time. For the second verifier, we use CPAchecker's value analysis with a time limit of 900 s.

Figure 3a uses a scatter plot to illustrate the CPU times of the reducer-based approach using Reducer (x-axis) against using Identity (i.e., pure sequential combination). The scatter plot shows results only for those verification tasks that at least one of the two combinations can solve and that none of them solved incorrectly or crashed on. Thus, the plot only displays results that have a useful result. Often, the results are similar (data points close to the diagonal). In this case, the predicate analysis alone already solved the verification task. For some tasks, Reducer is slower or even times out, due to the large size of residual programs. The reason is that Reducer restructures the program, e.g., unfolds loops and the program structure. The residual program becomes much larger and more complex in its structure, which complicates the task of the second verifier in these cases. However, there are also a set of tasks for which Reducer is significantly faster: the data points close to the upper border represent tasks for which the conditional combination with Reducer solved the task while the combination with Identity timed out. Thus, the null hypothesis is rejected.

Claim 1 (Comparison against native implementation). We compare our proposed reducer-based approach to construct conditional verifiers against the approach of the original implementation [10], which we refer to as 'native' approach because it implements the restriction of the state space according to the condition internally in the verifier. We use the same setup as above, but replace the second verifier by CPAchecker's value analysis with the internal condition treatment enabled. Figure 3b shows the useful results as scatter plot, again. Most of the data points are close to the diagonal, i.e., the two solutions perform similarly. However, as above, when the residual program gets too large, the reducer-based solution sometimes uses too much time (right side). For some tasks, the reducer-based solution is even faster than the native approach (top). Thus, Claim 1 is valid.

Claim 2 (Effective increase of number of verified programs). We now evaluate the claim that the use of two complementing verifiers joined by reducer-based conditional verification can effectively solve additional verification problems if additional resources are spent on running a combination after the runs of the original verifier. In our experiments, we always first run a conditional verifier based on predicate analysis to output the conditions, with a time limit of 100 s of CPU time. The predicate analysis combines lazy abstraction refinement [43] with predicate abstraction with adjustable-block encoding (ABE) [16]. ABE is configured to abstract

[6] https://github.com/sosy-lab/tbf
[7] https://github.com/sosy-lab/benchexec
[8] https://www.sosy-lab.org/research/reducer/

Table 2: Results of using a verifier on its own vs. a combination with predicate analysis and condition passing

	CPAseq	Smack	UAuto	Predicate +		
				CPAseq	Smack	UAuto
Correct	513	415	238	789	695	789
Correct proof	265	76	170	387	296	386
Correct alarm	248	339	68	402	399	403
Incorrect	0	0	7	0	0	4
Incorrect proof	0	0	4	0	0	0
Incorrect alarm	0	0	3	0	0	4
Unknown	307	405	575	31	125	27
Total	820	820	820	820	820	820

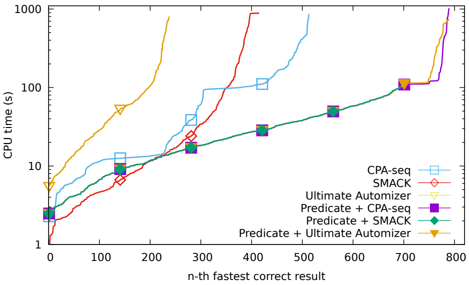


Figure 4: Quantile plots for the six verification approaches

at loop heads only. Let us refer to this verifier as **A**. The conditional verifier in the second verification step is always constructed from our reducer and an off-the-shelf verifier; we limit the CPU time to 900 s. Let us refer to this kind of verifier as **B**. Verifier **B** tries to solve all tasks that **A** was not yet able to solve, with the help of the conditions generated by **A** for these tasks. The time limit of 900 s of CPU time is considered a community standard (cf. SV-COMP), because most verification tasks can either be solved way below this time limit or cannot be solved at all. As verifier **B**, we compose our reducer with the three best tools from SV-COMP 2017 on reachability properties: CPAseq, Smack, and Ultimate Automizer.

Many of the verification tasks in the considered task set from the SV-COMP benchmarks are easy to solve for the standard verifiers. For those tasks, we do not need to further experiment because our aim is to show that the new approach can increase the overall number of verified programs. Therefore, we restrict our experiment to verification tasks that are hard-to-solve; in particular, we select those verification tasks for which at least one verifier **B** fails but the corresponding combination with condition passing of **A** and **B** solves the task. This results in a benchmark set containing 820 hard-to-solve verification tasks.

Table 2 breaks down the effectiveness of each verification approach. It lists the number of verification tasks that each verification approach solved correctly, solved incorrectly, and which it cannot solve ('Unknown'). The correct and incorrect results are further classified into answers that reported a proof and a bug, respectively. Inspecting the numbers, we observe the following: In all three cases, the reducer-based CMC combination with condition passing of verifiers **A** and **B** solves significantly more tasks correctly than verifier **B** alone. At the same time, the number of wrong answers is not increased by the conditional verifier. There are two possible reasons for this improvement: First, verifier **A** already accomplished the verification task, in which verifier **B** has no work (suggested by the data points on the diagonal with less than 100 s in Fig 3a). Or second, verifier **A** verified a significant portion of the verification task such that the residual program generated by Reducer becomes easier to analyze for verifier **B** (suggested by the middle and lower part of Table 3).

Figure 4 shows quantile plots for all six verification approaches. A data point (x, y) on such a graph means that the x fastest correct results can be solved all in max. y s of CPU time each. We observe that all reducer-based approaches significantly outperform their standalone counterpart by investing max. 100 s of CPU time. These observations together with Table 2 validate our Claim 2.

Claim 3 (Solving problems that none of the three can solve). We consider a particular subset of the verification tasks, namely those that none of the verifiers CPAseq, Smack, and Ultimate Automizer can solve as standard verifier but at least one combination can. These tasks seem to be particularly hard for verifiers while not being too hard for our approach. Table 3 shows an excerpt of those 143 programs of the task set. For each verification task (identified by name and expected verification result), the table contains groups of result, CPU time, and max. memory usage, for each of the three standard verifiers and their reducer-based combination with condition passing. From the table, it can be observed that Claim 3 is valid: there exist programs that conditional combinations can solve but none of the given standard verifiers can.

Claim 4 (Different back ends have different strengths). None of the conditional verifiers is superior. Each verifier has its strengths: for two verifiers **B** there exist verification tasks that only a combination with that verifier can solve and no other combination (cf. Table 2). And each verifier has its weaknesses: for each verifier, there are some verification tasks that the verifier, even in combination, cannot solve. To solve all difficult tasks, we need to leverage different technologies. The experimental results validate Claim 4.

This last observation makes the contribution of our reducer-based approach important: It does not make sense to extend existing verifiers to become conditional verifiers (in terms of accepting conditions as inputs), because we need *many* conditional verifiers. Our approach to take an *arbitrary* verifier off-the-shelf and construct a conditional verifier without implementation work significantly improves the overall achieved verification power.

Claim 5 (Reducer-based construction works also for testing). To demonstrate that our approach can be applied to tools other than model checkers, we combine our Reducer with three test-generation tools, namely AFL-fuzz (v2.46b), Crest (revision 31c32f4), and Klee (v1.4.0). As in the other experiments, the first analysis (which generates the condition) is the predicate analysis, again limited to 100 s. The test-generation is limited to 900 s.

Analogous to Claim 2, we restrict the experiment to those verification tasks that are hard-to-solve with test generation: we select those tasks for which at least one test generator fails to uncover a bug in, but that the corresponding combination with condition passing can correctly solve. In addition, since testing cannot prove correctness, we only consider verification tasks that are unsafe.

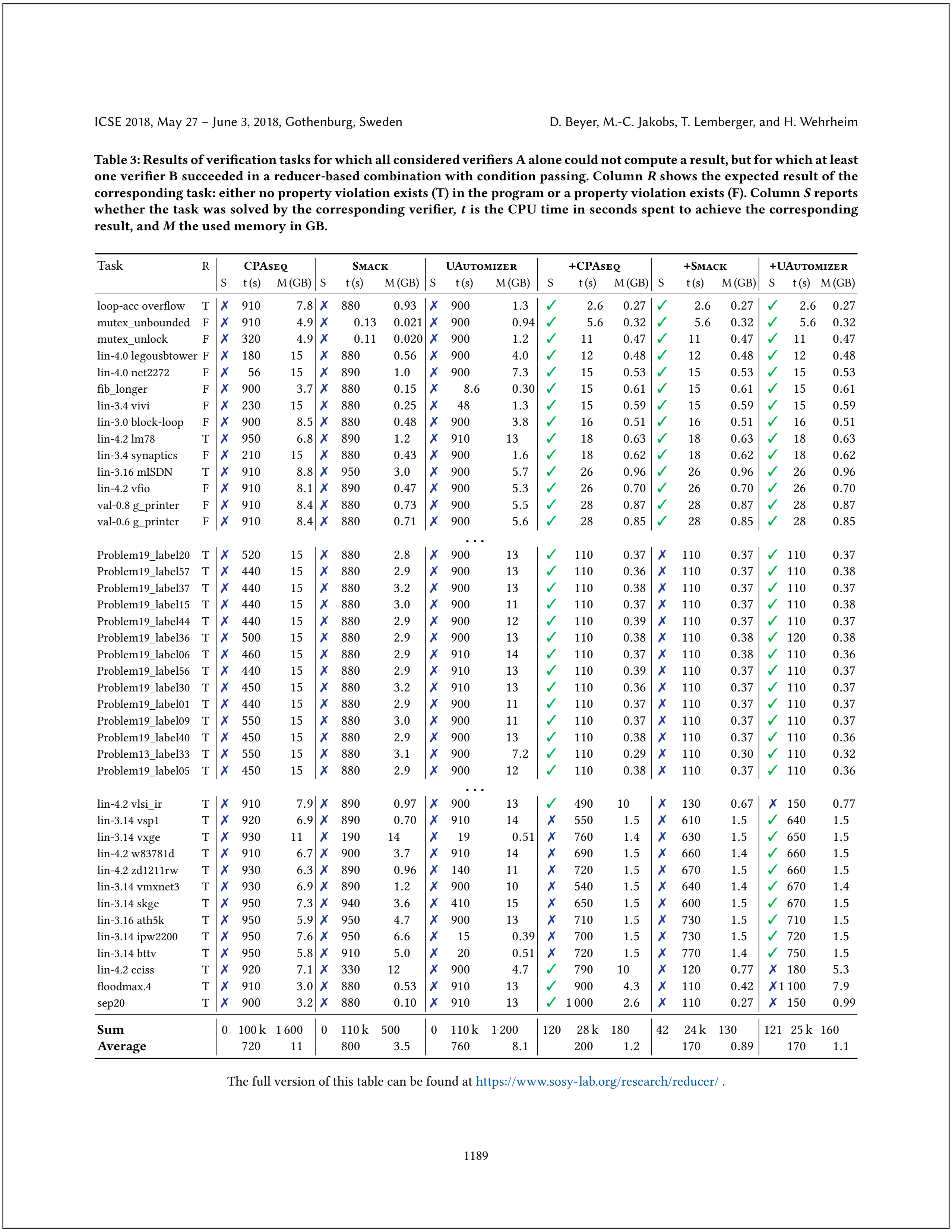


Table 3: Results of verification tasks for which all considered verifiers A alone could not compute a result, but for which at least one verifier B succeeded in a reducer-based combination with condition passing. Column *R* shows the expected result of the corresponding task: either no property violation exists (T) in the program or a property violation exists (F). Column *S* reports whether the task was solved by the corresponding verifier, *t* is the CPU time in seconds spent to achieve the corresponding result, and *M* the used memory in GB.

Task	R	CPAseq			Smack			UAutomizer			+CPAseq			+Smack			+UAutomizer		
		S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)	S	t (s)	M (GB)
loop-acc overflow	T	✗	910	7.8	✗	880	0.93	✗	900	1.3	✓	2.6	0.27	✓	2.6	0.27	✓	2.6	0.27
mutex_unbounded	F	✗	910	4.9	✗	0.13	0.021	✗	900	0.94	✓	5.6	0.32	✓	5.6	0.32	✓	5.6	0.32
mutex_unlock	F	✗	320	4.9	✗	0.11	0.020	✗	900	1.2	✓	11	0.47	✓	11	0.47	✓	11	0.47
lin-4.0 legousbtower	F	✗	180	15	✗	880	0.56	✗	900	4.0	✓	12	0.48	✓	12	0.48	✓	12	0.48
lin-4.0 net2272	F	✗	56	15	✗	890	1.0	✗	900	7.3	✓	15	0.53	✓	15	0.53	✓	15	0.53
fib_longer	F	✗	900	3.7	✗	880	0.15	✗	8.6	0.30	✓	15	0.61	✓	15	0.61	✓	15	0.61
lin-3.4 vivi	F	✗	230	15	✗	880	0.25	✗	48	1.3	✓	15	0.59	✓	15	0.59	✓	15	0.59
lin-3.0 block-loop	F	✗	900	8.5	✗	880	0.48	✗	900	3.8	✓	16	0.51	✓	16	0.51	✓	16	0.51
lin-4.2 lm78	T	✗	950	6.8	✗	890	1.2	✗	910	13	✓	18	0.63	✓	18	0.63	✓	18	0.63
lin-3.4 synaptics	F	✗	210	15	✗	880	0.43	✗	900	1.6	✓	18	0.62	✓	18	0.62	✓	18	0.62
lin-3.16 mISDN	T	✗	910	8.8	✗	950	3.0	✗	900	5.7	✓	26	0.96	✓	26	0.96	✓	26	0.96
lin-4.2 vfio	F	✗	910	8.1	✗	890	0.47	✗	900	5.3	✓	26	0.70	✓	26	0.70	✓	26	0.70
val-0.8 g_printer	F	✗	910	8.4	✗	880	0.73	✗	900	5.5	✓	28	0.87	✓	28	0.87	✓	28	0.87
val-0.6 g_printer	F	✗	910	8.4	✗	880	0.71	✗	900	5.6	✓	28	0.85	✓	28	0.85	✓	28	0.85
									…										
Problem19_label20	T	✗	520	15	✗	880	2.8	✗	900	13	✓	110	0.37	✗	110	0.37	✓	110	0.37
Problem19_label57	T	✗	440	15	✗	880	2.9	✗	900	13	✓	110	0.36	✗	110	0.37	✓	110	0.38
Problem19_label37	T	✗	440	15	✗	880	3.2	✗	900	13	✓	110	0.38	✗	110	0.37	✓	110	0.37
Problem19_label15	T	✗	440	15	✗	880	3.0	✗	900	11	✓	110	0.37	✗	110	0.37	✓	110	0.38
Problem19_label44	T	✗	440	15	✗	880	2.9	✗	900	12	✓	110	0.39	✗	110	0.37	✓	110	0.37
Problem19_label36	T	✗	500	15	✗	880	2.9	✗	900	13	✓	110	0.38	✗	110	0.38	✓	120	0.38
Problem19_label06	T	✗	460	15	✗	880	2.9	✗	910	14	✓	110	0.37	✗	110	0.38	✓	110	0.36
Problem19_label56	T	✗	440	15	✗	880	2.9	✗	910	13	✓	110	0.39	✗	110	0.37	✓	110	0.37
Problem19_label30	T	✗	450	15	✗	880	3.2	✗	910	13	✓	110	0.36	✗	110	0.37	✓	110	0.37
Problem19_label01	T	✗	440	15	✗	880	2.9	✗	900	11	✓	110	0.37	✗	110	0.37	✓	110	0.37
Problem19_label09	T	✗	550	15	✗	880	3.0	✗	900	11	✓	110	0.37	✗	110	0.37	✓	110	0.37
Problem19_label40	T	✗	450	15	✗	880	2.9	✗	900	13	✓	110	0.38	✗	110	0.37	✓	110	0.36
Problem13_label33	T	✗	550	15	✗	880	3.1	✗	900	7.2	✓	110	0.29	✗	110	0.30	✓	110	0.32
Problem19_label05	T	✗	450	15	✗	880	2.9	✗	900	12	✓	110	0.38	✗	110	0.37	✓	110	0.36
									…										
lin-4.2 vlsi_ir	T	✗	910	7.9	✗	890	0.97	✗	900	13	✓	490	10	✗	130	0.67	✗	150	0.77
lin-3.14 vsp1	T	✗	920	6.9	✗	890	0.70	✗	910	14	✗	550	1.5	✗	610	1.5	✓	640	1.5
lin-3.14 vxge	T	✗	930	11	✗	190	14	✗	19	0.51	✗	760	1.4	✗	630	1.5	✓	650	1.5
lin-4.2 w83781d	T	✗	910	6.7	✗	900	3.7	✗	910	14	✗	690	1.5	✗	660	1.4	✓	660	1.5
lin-4.2 zd1211rw	T	✗	930	6.3	✗	890	0.96	✗	140	11	✗	720	1.5	✗	670	1.5	✓	660	1.5
lin-3.14 vmxnet3	T	✗	930	6.9	✗	890	1.2	✗	900	10	✗	540	1.5	✗	640	1.4	✓	670	1.4
lin-3.14 skge	T	✗	950	7.3	✗	940	3.6	✗	410	15	✗	650	1.5	✗	600	1.5	✓	670	1.5
lin-3.16 ath5k	T	✗	950	5.9	✗	950	4.7	✗	900	13	✗	710	1.5	✗	730	1.5	✓	710	1.5
lin-3.14 ipw2200	T	✗	950	7.6	✗	950	6.6	✗	15	0.39	✗	700	1.5	✗	730	1.5	✓	720	1.5
lin-3.14 bttv	T	✗	950	5.8	✗	910	5.0	✗	20	0.51	✗	720	1.5	✗	770	1.4	✓	750	1.5
lin-4.2 cciss	T	✗	920	7.1	✗	330	12	✗	900	4.7	✓	790	10	✗	120	0.77	✗	180	5.3
floodmax.4	T	✗	910	3.0	✗	880	0.53	✗	910	13	✓	900	4.3	✗	110	0.42	✗	1 100	7.9
sep20	T	✗	900	3.2	✗	880	0.10	✗	910	13	✓	1 000	2.6	✗	110	0.27	✗	150	0.99
Sum		0	100 k	1 600	0	110 k	500	0	110 k	1 200	120	28 k	180	42	24 k	130	121	25 k	160
Average			720	11		800	3.5		760	8.1		200	1.2		170	0.89		170	1.1

The full version of this table can be found at https://www.sosy-lab.org/research/reducer/ .

Table 4: Test generation vs. CMC combination

	AFL-fuzz	Crest	Klee	Predicate +		
				AFL-fuzz	Crest	Klee
Correct alarm	96	44	277	479	476	477
Incorrect proof	0	0	0	0	0	0
Unknown	384	436	203	1	4	3
Total	480	480	480	480	480	480

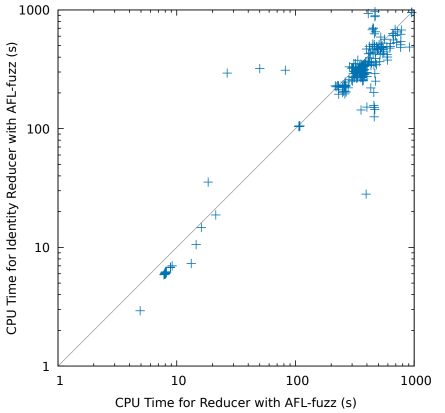


Figure 5: CPU time for predicate analysis and AFL-fuzz combined with Reducer (CMC) and with Identity (sequential)

As a result, we get 480 tasks. Table 4 compares the performance of the CMC scenarios with the tester performance. Similar to Table 2, it shows the numbers of correct alarms, incorrect proofs, and unsolved tasks. However, it leaves out the rows related to safe verification tasks. We see that for all three test-generation tools the number of correct alarms of our reducer-based combination with condition passing is higher than for the respective tester. In general, such an improvement is not only caused by the use of the verifier **A**, but often a result of the combination of tools. To further support this statement, we present Fig. 5. It shows the CPU time of two reducer-based CMC solutions, both using the predicate analysis mentioned above to generate conditions, using the full set of 1 501 verification tasks with expected result *false*. The first solution (x-axis) uses the reducer Reducer with AFL-fuzz and the second solution (y-axis) uses the Identity reducer with AFL-fuzz (pure sequential combination). For better visualization, we removed the results that the predicate analysis can solve on its own. Due to the mentioned blowup of the residual program, the Reducer based solution (Reducer plus AFL-fuzz) performs worse for some tasks, but it can also solve a significant amount of tasks faster than the pure sequential combination (Identity plus AFL-fuzz).

Size of residual programs. As already mentioned, the residual program created by Reducer may become significantly larger than the original program. The reason is a large amount of branching in the condition, i.e., unfolding of loops and program structure, which is needed to record the verification work already performed. To study this in more detail, we compared the sizes of the original and the residual program in terms of locations in the CFA. At worst, the residual program was more than 10 times larger than the original program (1 934 vs. 22 325 locations). At best, the number of locations in the residual program is less than 1 % of the number of original program locations (200 253 vs. 127 locations). On average, the residual program contains fewer locations (with a mean of 27 % and a median of 14 % of the number of locations in the original program). While the residual program can be much larger, it is often much smaller.

4.4 Threats to Validity

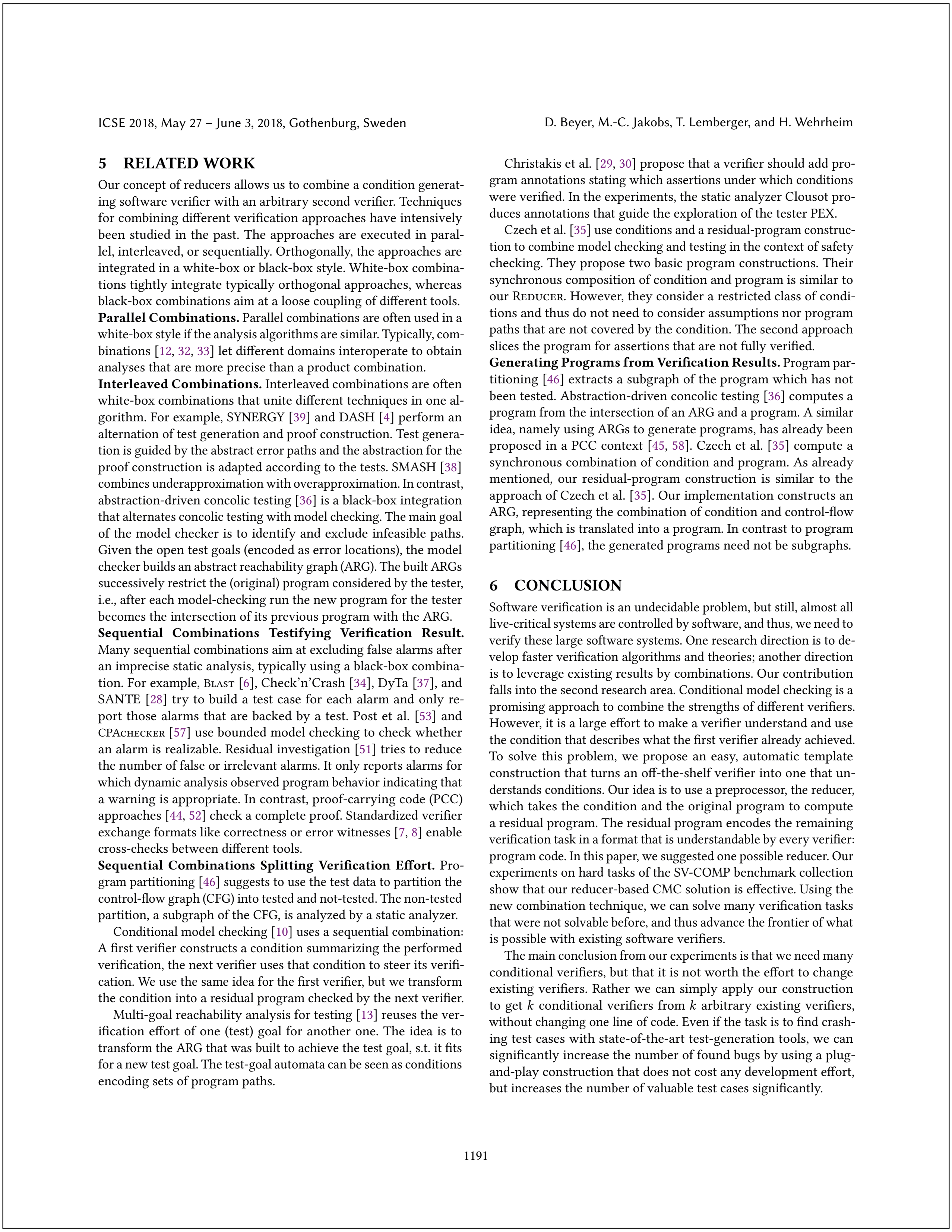
We did not cross-check the reported verification results with an independent verifier because we currently do not know how to construct correctness or violation witnesses [7, 8] in the setting of reducer-based conditional model checking. While we are sure that the standalone verifiers did a proper inspection (they successfully participated in SV-COMP or provide a test), tools might have guessed the correct answer when run as part of the conditional verifier. Yet, we think that guessing is unlikely. The tools are laid out to provide witnesses and thus properly perform their verification.

The correctness of the residual program is another threat. Like other analysis tools, we rely on the soundness of the transformation from program to CFA and back. Additionally, we rely on the soundness of the existing condition generating tool in that the condition only covers paths the verifier has already inspected. Furthermore, our implementation of Reducer is a prototype which revealed bugs during evaluation. In principle, the bugs might be the reason for the effectiveness of the reducer-based approach. However, the bugs we observed led the conditional verifier to report a wrong result.

Additionally, we checked the null hypothesis and claim 1 only with a single condition generating analysis and a single conditional verifier. Thus, the corresponding results might not be universally valid in any reducer-based conditional-model-checking setup.

When using a combination of two verification tools with CMC, it is also possible that the increase in solvable tasks is simply because the different tools can solve a distinct set of tasks each in a very short time. E.g., in our configuration, it could have been possible that the full increase in additionally solvable tasks is only due to a competence of the predicate analysis in quickly solving a set of tasks that none of the other verifiers can solve. To make sure that our considered tool combinations actually benefit from the use of condition passing, we provided a comparison with a pure sequential combination (Identity) that showed the general benefit. In addition, CPAseq includes a 200 s run of predicate analysis in its configuration—this ensures that all benefits for our combination of CPAseq and predicate analysis are actually due to our Reducer approach. Of the 820 tasks considered in the experiments backing claim 2, 143 cannot be solved by any of the sequential combinations, but only when using CMC with condition passing.

We only consider a subset of the SV-COMP tasks and the three best verifiers from SV-COMP. These three verifiers might be tuned to SV-COMP tasks and may perform worse on our generated residual programs. Despite this possible bias, our approach still improves the existing verifiers. In addition, the three test-generation tools used never participated in any edition of SV-COMP and are unlikely biased. Our approach still shows improvements for them.



5 RELATED WORK

Our concept of reducers allows us to combine a condition generating software verifier with an arbitrary second verifier. Techniques for combining different verification approaches have intensively been studied in the past. The approaches are executed in parallel, interleaved, or sequentially. Orthogonally, the approaches are integrated in a white-box or black-box style. White-box combinations tightly integrate typically orthogonal approaches, whereas black-box combinations aim at a loose coupling of different tools.

Parallel Combinations. Parallel combinations are often used in a white-box style if the analysis algorithms are similar. Typically, combinations [12, 32, 33] let different domains interoperate to obtain analyses that are more precise than a product combination.

Interleaved Combinations. Interleaved combinations are often white-box combinations that unite different techniques in one algorithm. For example, SYNERGY [39] and DASH [4] perform an alternation of test generation and proof construction. Test generation is guided by the abstract error paths and the abstraction for the proof construction is adapted according to the tests. SMASH [38] combines underapproximation with overapproximation. In contrast, abstraction-driven concolic testing [36] is a black-box integration that alternates concolic testing with model checking. The main goal of the model checker is to identify and exclude infeasible paths. Given the open test goals (encoded as error locations), the model checker builds an abstract reachability graph (ARG). The built ARGs successively restrict the (original) program considered by the tester, i.e., after each model-checking run the new program for the tester becomes the intersection of its previous program with the ARG.

Sequential Combinations Testifying Verification Result. Many sequential combinations aim at excluding false alarms after an imprecise static analysis, typically using a black-box combination. For example, Blast [6], Check'n'Crash [34], DyTa [37], and SANTE [28] try to build a test case for each alarm and only report those alarms that are backed by a test. Post et al. [53] and CPAchecker [57] use bounded model checking to check whether an alarm is realizable. Residual investigation [51] tries to reduce the number of false or irrelevant alarms. It only reports alarms for which dynamic analysis observed program behavior indicating that a warning is appropriate. In contrast, proof-carrying code (PCC) approaches [44, 52] check a complete proof. Standardized verifier exchange formats like correctness or error witnesses [7, 8] enable cross-checks between different tools.

Sequential Combinations Splitting Verification Effort. Program partitioning [46] suggests to use the test data to partition the control-flow graph (CFG) into tested and not-tested. The non-tested partition, a subgraph of the CFG, is analyzed by a static analyzer.

Conditional model checking [10] uses a sequential combination: A first verifier constructs a condition summarizing the performed verification, the next verifier uses that condition to steer its verification. We use the same idea for the first verifier, but we transform the condition into a residual program checked by the next verifier.

Multi-goal reachability analysis for testing [13] reuses the verification effort of one (test) goal for another one. The idea is to transform the ARG that was built to achieve the test goal, s.t. it fits for a new test goal. The test-goal automata can be seen as conditions encoding sets of program paths.

Christakis et al. [29, 30] propose that a verifier should add program annotations stating which assertions under which conditions were verified. In the experiments, the static analyzer Clousot produces annotations that guide the exploration of the tester PEX.

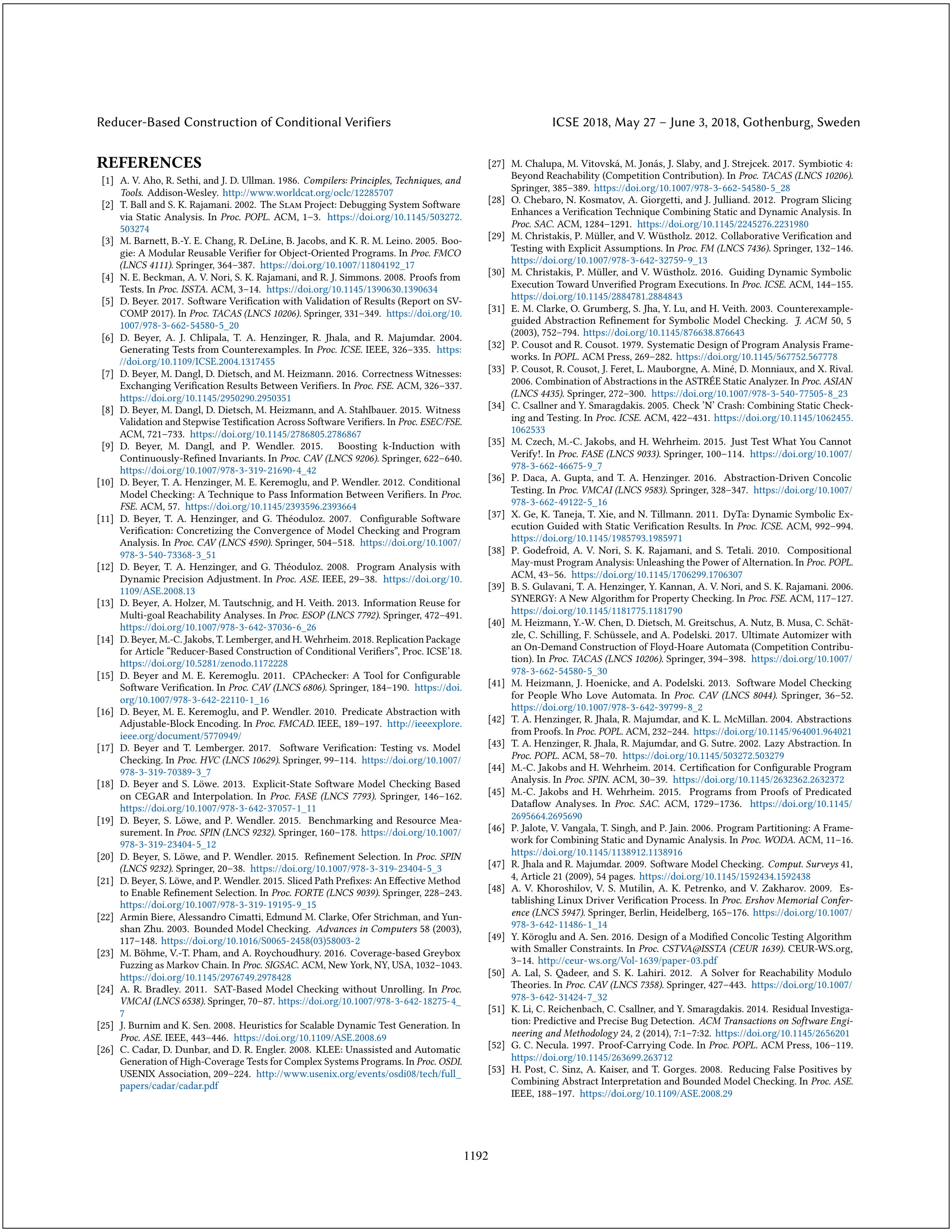
Czech et al. [35] use conditions and a residual-program construction to combine model checking and testing in the context of safety checking. They propose two basic program constructions. Their synchronous composition of condition and program is similar to our Reducer. However, they consider a restricted class of conditions and thus do not need to consider assumptions nor program paths that are not covered by the condition. The second approach slices the program for assertions that are not fully verified.

Generating Programs from Verification Results. Program partitioning [46] extracts a subgraph of the program which has not been tested. Abstraction-driven concolic testing [36] computes a program from the intersection of an ARG and a program. A similar idea, namely using ARGs to generate programs, has already been proposed in a PCC context [45, 58]. Czech et al. [35] compute a synchronous combination of condition and program. As already mentioned, our residual-program construction is similar to the approach of Czech et al. [35]. Our implementation constructs an ARG, representing the combination of condition and control-flow graph, which is translated into a program. In contrast to program partitioning [46], the generated programs need not be subgraphs.

6 CONCLUSION

Software verification is an undecidable problem, but still, almost all live-critical systems are controlled by software, and thus, we need to verify these large software systems. One research direction is to develop faster verification algorithms and theories; another direction is to leverage existing results by combinations. Our contribution falls into the second research area. Conditional model checking is a promising approach to combine the strengths of different verifiers. However, it is a large effort to make a verifier understand and use the condition that describes what the first verifier already achieved. To solve this problem, we propose an easy, automatic template construction that turns an off-the-shelf verifier into one that understands conditions. Our idea is to use a preprocessor, the reducer, which takes the condition and the original program to compute a residual program. The residual program encodes the remaining verification task in a format that is understandable by every verifier: program code. In this paper, we suggested one possible reducer. Our experiments on hard tasks of the SV-COMP benchmark collection show that our reducer-based CMC solution is effective. Using the new combination technique, we can solve many verification tasks that were not solvable before, and thus advance the frontier of what is possible with existing software verifiers.

The main conclusion from our experiments is that we need many conditional verifiers, but that it is not worth the effort to change existing verifiers. Rather we can simply apply our construction to get k conditional verifiers from k arbitrary existing verifiers, without changing one line of code. Even if the task is to find crashing test cases with state-of-the-art test-generation tools, we can significantly increase the number of found bugs by using a plug-and-play construction that does not cost any development effort, but increases the number of valuable test cases significantly.



REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley. http://www.worldcat.org/oclc/12285707

[2] T. Ball and S. K. Rajamani. 2002. The Slam Project: Debugging System Software via Static Analysis. In *Proc. POPL*. ACM, 1–3. https://doi.org/10.1145/503272.503274

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proc. FMCO (LNCS 4111)*. Springer, 364–387. https://doi.org/10.1007/11804192_17

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. 2008. Proofs from Tests. In *Proc. ISSTA*. ACM, 3–14. https://doi.org/10.1145/1390630.1390634

[5] D. Beyer. 2017. Software Verification with Validation of Results (Report on SV-COMP 2017). In *Proc. TACAS (LNCS 10206)*. Springer, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating Tests from Counterexamples. In *Proc. ICSE*. IEEE, 326–335. https://doi.org/10.1109/ICSE.2004.1317455

[7] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

[8] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification Across Software Verifiers. In *Proc. ESEC/FSE*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867

[9] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-Induction with Continuously-Refined Invariants. In *Proc. CAV (LNCS 9206)*. Springer, 622–640. https://doi.org/10.1007/978-3-319-21690-4_42

[10] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional Model Checking: A Technique to Pass Information Between Verifiers. In *Proc. FSE*. ACM, 57. https://doi.org/10.1145/2393596.2393664

[11] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[12] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2008. Program Analysis with Dynamic Precision Adjustment. In *Proc. ASE*. IEEE, 29–38. https://doi.org/10.1109/ASE.2008.13

[13] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. 2013. Information Reuse for Multi-goal Reachability Analyses. In *Proc. ESOP (LNCS 7792)*. Springer, 472–491. https://doi.org/10.1007/978-3-642-37036-6_26

[14] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. 2018. Replication Package for Article "Reducer-Based Construction of Conditional Verifiers", Proc. ICSE'18. https://doi.org/10.5281/zenodo.1172228

[15] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proc. CAV (LNCS 6806)*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[16] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. IEEE, 189–197. http://ieeexplore.ieee.org/document/5770949/

[17] D. Beyer and T. Lemberger. 2017. Software Verification: Testing vs. Model Checking. In *Proc. HVC (LNCS 10629)*. Springer, 99–114. https://doi.org/10.1007/978-3-319-70389-3_7

[18] D. Beyer and S. Löwe. 2013. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE (LNCS 7793)*. Springer, 146–162. https://doi.org/10.1007/978-3-642-37057-1_11

[19] D. Beyer, S. Löwe, and P. Wendler. 2015. Benchmarking and Resource Measurement. In *Proc. SPIN (LNCS 9232)*. Springer, 160–178. https://doi.org/10.1007/978-3-319-23404-5_12

[20] D. Beyer, S. Löwe, and P. Wendler. 2015. Refinement Selection. In *Proc. SPIN (LNCS 9232)*. Springer, 20–38. https://doi.org/10.1007/978-3-319-23404-5_3

[21] D. Beyer, S. Löwe, and P. Wendler. 2015. Sliced Path Prefixes: An Effective Method to Enable Refinement Selection. In *Proc. FORTE (LNCS 9039)*. Springer, 228–243. https://doi.org/10.1007/978-3-319-19195-9_15

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded Model Checking. *Advances in Computers* 58 (2003), 117–148. https://doi.org/10.1016/S0065-2458(03)58003-2

[23] M. Böhme, V.-T. Pham, and A. Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proc. SIGSAC*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[24] A. R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Proc. VMCAI (LNCS 6538)*. Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[25] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proc. ASE*. IEEE, 443–446. https://doi.org/10.1109/ASE.2008.69

[26] C. Cadar, D. Dunbar, and D. R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI*. USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[27] M. Chalupa, M. Vitovská, M. Jonás, J. Slaby, and J. Strejcek. 2017. Symbiotic 4: Beyond Reachability (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 385–389. https://doi.org/10.1007/978-3-662-54580-5_28

[28] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. 2012. Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis. In *Proc. SAC*. ACM, 1284–1291. https://doi.org/10.1145/2245276.2231980

[29] M. Christakis, P. Müller, and V. Wüstholz. 2012. Collaborative Verification and Testing with Explicit Assumptions. In *Proc. FM (LNCS 7436)*. Springer, 132–146. https://doi.org/10.1007/978-3-642-32759-9_13

[30] M. Christakis, P. Müller, and V. Wüstholz. 2016. Guiding Dynamic Symbolic Execution Toward Unverified Program Executions. In *Proc. ICSE*. ACM, 144–155. https://doi.org/10.1145/2884781.2884843

[31] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[32] P. Cousot and R. Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM Press, 269–282. https://doi.org/10.1145/567752.567778

[33] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2006. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Proc. ASIAN (LNCS 4435)*. Springer, 272–300. https://doi.org/10.1007/978-3-540-77505-8_23

[34] C. Csallner and Y. Smaragdakis. 2005. Check 'N' Crash: Combining Static Checking and Testing. In *Proc. ICSE*. ACM, 422–431. https://doi.org/10.1145/1062455.1062533

[35] M. Czech, M.-C. Jakobs, and H. Wehrheim. 2015. Just Test What You Cannot Verify!. In *Proc. FASE (LNCS 9033)*. Springer, 100–114. https://doi.org/10.1007/978-3-662-46675-9_7

[36] P. Daca, A. Gupta, and T. A. Henzinger. 2016. Abstraction-Driven Concolic Testing. In *Proc. VMCAI (LNCS 9583)*. Springer, 328–347. https://doi.org/10.1007/978-3-662-49122-5_16

[37] X. Ge, K. Taneja, T. Xie, and N. Tillmann. 2011. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results. In *Proc. ICSE*. ACM, 992–994. https://doi.org/10.1145/1985793.1985971

[38] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. 2010. Compositional May-must Program Analysis: Unleashing the Power of Alternation. In *Proc. POPL*. ACM, 43–56. https://doi.org/10.1145/1706299.1706307

[39] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. 2006. SYNERGY: A New Algorithm for Property Checking. In *Proc. FSE*. ACM, 117–127. https://doi.org/10.1145/1181775.1181790

[40] M. Heizmann, Y.-W. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. 2017. Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 394–398. https://doi.org/10.1007/978-3-662-54580-5_30

[41] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[42] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. 2004. Abstractions from Proofs. In *Proc. POPL*. ACM, 232–244. https://doi.org/10.1145/964001.964021

[43] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. 2002. Lazy Abstraction. In *Proc. POPL*. ACM, 58–70. https://doi.org/10.1145/503272.503279

[44] M.-C. Jakobs and H. Wehrheim. 2014. Certification for Configurable Program Analysis. In *Proc. SPIN*. ACM, 30–39. https://doi.org/10.1145/2632362.2632372

[45] M.-C. Jakobs and H. Wehrheim. 2015. Programs from Proofs of Predicated Dataflow Analyses. In *Proc. SAC*. ACM, 1729–1736. https://doi.org/10.1145/2695664.2695690

[46] P. Jalote, V. Vangala, T. Singh, and P. Jain. 2006. Program Partitioning: A Framework for Combining Static and Dynamic Analysis. In *Proc. WODA*. ACM, 11–16. https://doi.org/10.1145/1138912.1138916

[47] R. Jhala and R. Majumdar. 2009. Software Model Checking. *Comput. Surveys* 41, 4, Article 21 (2009), 54 pages. https://doi.org/10.1145/1592434.1592438

[48] A. V. Khoroshilov, V. S. Mutilin, A. K. Petrenko, and V. Zakharov. 2009. Establishing Linux Driver Verification Process. In *Proc. Ershov Memorial Conference (LNCS 5947)*. Springer, Berlin, Heidelberg, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14

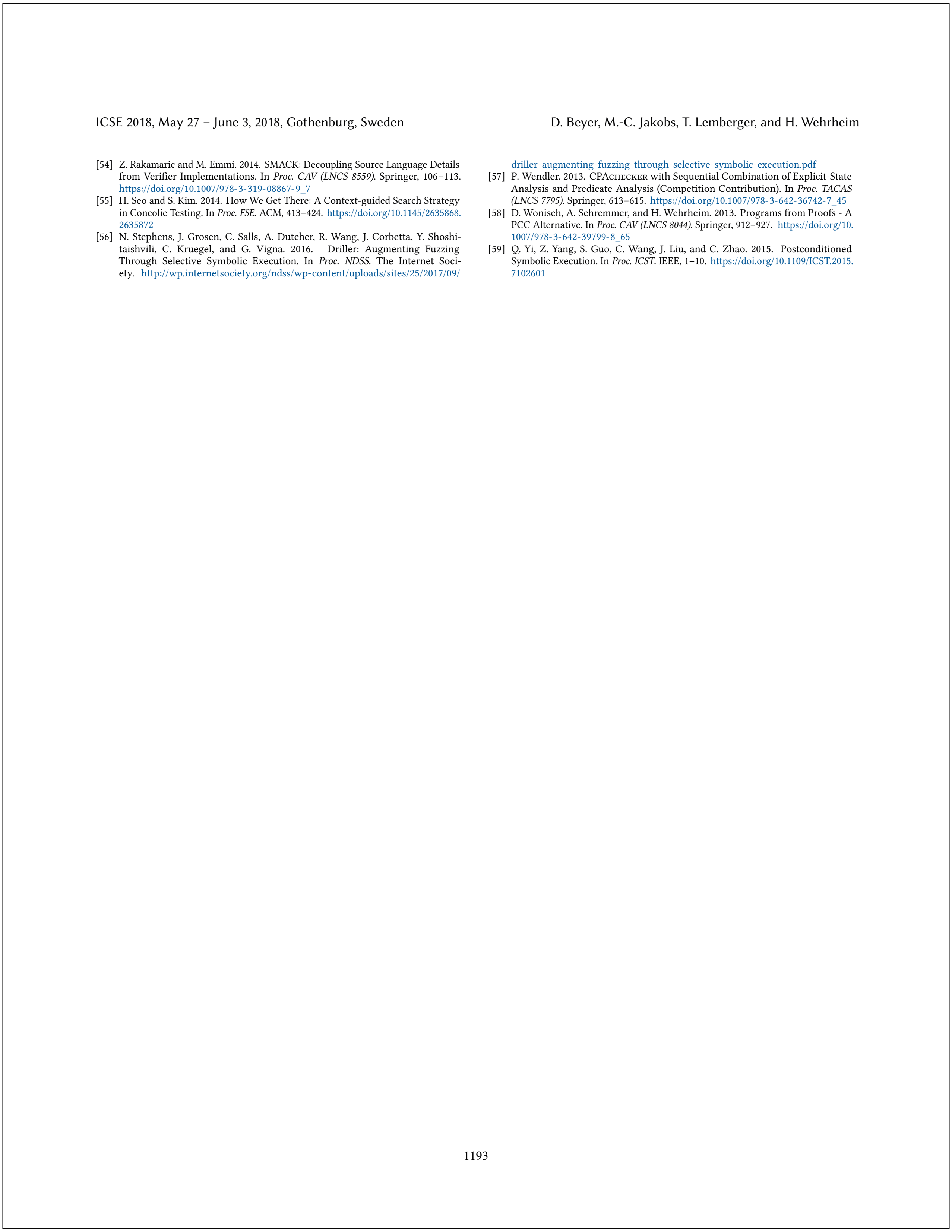
[49] Y. Köroglu and A. Sen. 2016. Design of a Modified Concolic Testing Algorithm with Smaller Constraints. In *Proc. CSTVA@ISSTA (CEUR 1639)*. CEUR-WS.org, 3–14. http://ceur-ws.org/Vol-1639/paper-03.pdf

[50] A. Lal, S. Qadeer, and S. K. Lahiri. 2012. A Solver for Reachability Modulo Theories. In *Proc. CAV (LNCS 7358)*. Springer, 427–443. https://doi.org/10.1007/978-3-642-31424-7_32

[51] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. 2014. Residual Investigation: Predictive and Precise Bug Detection. *ACM Transactions on Software Engineering and Methodology* 24, 2 (2014), 7:1–7:32. https://doi.org/10.1145/2656201

[52] G. C. Necula. 1997. Proof-Carrying Code. In *Proc. POPL*. ACM Press, 106–119. https://doi.org/10.1145/263699.263712

[53] H. Post, C. Sinz, A. Kaiser, and T. Gorges. 2008. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. In *Proc. ASE*. IEEE, 188–197. https://doi.org/10.1109/ASE.2008.29



[54] Z. Rakamaric and M. Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proc. CAV (LNCS 8559)*. Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[55] H. Seo and S. Kim. 2014. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proc. FSE*. ACM, 413–424. https://doi.org/10.1145/2635868.2635872

[56] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. NDSS*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[57] P. Wendler. 2013. CPAchecker with Sequential Combination of Explicit-State Analysis and Predicate Analysis (Competition Contribution). In *Proc. TACAS (LNCS 7795)*. Springer, 613–615. https://doi.org/10.1007/978-3-642-36742-7_45

[58] D. Wonisch, A. Schremmer, and H. Wehrheim. 2013. Programs from Proofs - A PCC Alternative. In *Proc. CAV (LNCS 8044)*. Springer, 912–927. https://doi.org/10.1007/978-3-642-39799-8_65

[59] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao. 2015. Postconditioned Symbolic Execution. In *Proc. ICST*. IEEE, 1–10. https://doi.org/10.1109/ICST.2015.7102601

Conditional Testing

Off-the-Shelf Combination of Test-Case Generators

Dirk Beyer and Thomas Lemberger

LMU Munich, Munich, Germany

Abstract. There are several powerful automatic testers available, each with different strengths and weaknesses. To immediately benefit from different strengths of different tools, we need to investigate ways for quick and easy combination of techniques. Until now, research has mostly investigated integrated combinations, which require extra implementation effort. We propose the concept of *conditional testing* and a set of combination techniques that do not require implementation effort: Different testers can be taken 'off the shelf' and combined in a way that they *cooperatively* solve the problem of test-case generation for a given input program and coverage criterion. This way, the latest advances in test-case generation can be combined without delay. Conditional testing passes the test goals that a first tester has covered to the next tester, so that the next tester does not need to repeat work (as in combinations without information passing) but can focus on the remaining test goals. Our combinations do not require changes to the implementation of a tester, because we leverage a testability transformation (i.e., we reduce the input program to those parts that are relevant to the remaining test goals). To evaluate conditional testing and our proposed combination techniques, we (1) implemented the generic conditional tester CondTest, including the required transformations, and (2) ran experiments on a large amount of benchmark tasks; the obtained results are promising.

Keywords: Software testing · Test-case generation · Conditional model checking · Cooperative verification · Software verification · Program analysis · Test coverage

1 Introduction

Tool competitions in software verification and testing [1,2,26,34] have shown that there is no tool that is superior, but that different tools and approaches have different strengths. Therefore, we need to combine different tools and approaches. Integrated combination approaches [8,15,19,22] have shown their potential, but those combinations require additional implementation work.

The goal of this paper is to provide a generic framework that enables combinations of tools for test-case generation without the need to change the tools: We show how to take a set of testers 'off the shelf' and combine them on the

Supported in part by DFG grant BE 1761/7-1.

© Springer Nature Switzerland AG 2019

Y.-F. Chen et al. (Eds.): ATVA 2019, LNCS 11781, pp. 189–208, 2019.
https://doi.org/10.1007/978-3-030-31784-3_11

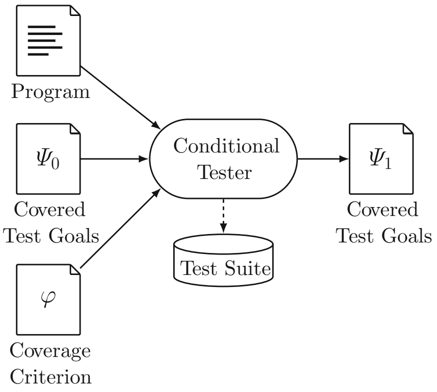


Fig. 1. Conditional testing

```
int main() {
 int x = input();
 if (x != 161) {
  // ...
 } else {
  // ...
 }
}
```

Fig. 2. Program under test

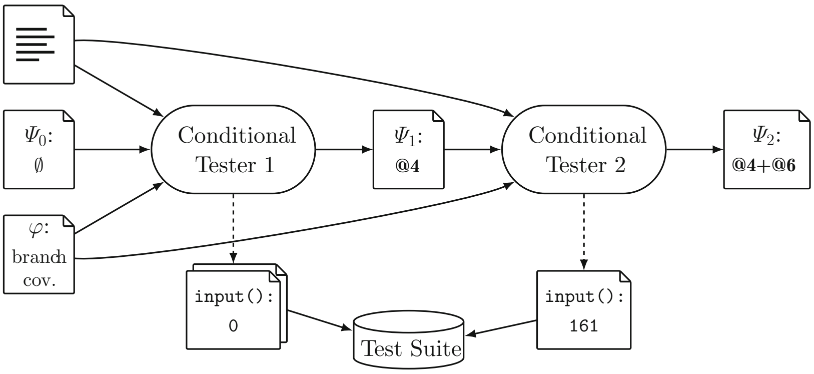


Fig. 3. Example usage of conditional testers

binary level. In other words, any tester can be taken as a black box, wrapped into a new meta tester (conditional tester) by a fully automated construction, and the new tester uses interfaces that make it possible to combine it with others (for an overview of combination techniques for software verification see [12]). There are several successful testers for C programs already; nine of them participated in the competition on software testing [2] and adhere to standard exchange formats for their input and output.

Conditional testing applies the idea of conditional model checking [7] to testing, as illustrated in Fig. 1. A *conditional tester* gets as input a program under test, a coverage criterion φ (e.g., ‘branch coverage’), and a condition Ψ_0 that describes a set of test goals that are already covered by existing tests. With this information, a conditional tester creates (1) a test suite that tries to cover as many test goals of $[\![\varphi]\!] \setminus \Psi_0$ as possible, and (2) a new condition Ψ_1 of test goals that have been covered. For a coverage criterion φ, we use $[\![\varphi]\!]$ to denote the test goals that are needed to fulfill φ. The condition Ψ_1 covers the test goals described by condition Ψ_0 and the test goals newly covered by the created test suite. With this interface for information passing, conditional testers can be combined to focus on different or remaining test goals.

Figure 2 shows a small program that we use to illustrate conditional testing. The program gets an arbitrary integer as input and stores it in program variable `x`. The program then checks whether `x` is un-equal to value 161. If it is, the if-branch is entered and some more code (`// ...`) is executed. Otherwise, the else-branch is entered. The coverage criterion of branch coverage defines two test goals for this program: (1) cover line 4 (denoted by @4 in FQL [25]), and (2) cover line 6 (@6). A test suite that covers both test goals would contain at least two test cases: One with input 161, and one with any other input. A randomized tester can quickly generate a test case with an input different from 161, because the number of possible values is very high and thus very probable to be fulfilled by a random test case. In contrast, it is difficult for a randomized tester to create a test case with input 161.

Let us consider the combination of a fast, but shallow randomized tester (Conditional Tester 1 in Fig. 3) with a tester that is slower, but uses an exhaustive reasoning technique (Conditional Tester 2), to obtain a test suite that covers all branches: Given the program under test, the coverage criterion φ = COVER EDGES(@DECISIONEDGE) (branch coverage in FQL syntax), and the empty condition $\Psi_0 = \emptyset$ (i.e., no test goal covered yet), we run the conditional, random tester for a short amount of time. Assume it creates several different test cases, including one with input value 0. The created test cases cover line 4, but do not cover line 6. Thus, the conditional tester returns Ψ_1 = **@4** for the now covered test goal. A second conditional tester, for example based on symbolic execution, then gets the same program and coverage criterion, but condition Ψ_1. The conditional tester focuses on the remaining test goal of covering line 6 and creates a test case with input 161. Now, all test goals are covered and Ψ_2 = **@4**+**@6** describes both test goals.

Conditional testing does not prescribe a certain format or language to be used for specifying coverage criteria and conditions. The competition on software testing [2] uses FQL [24,25] as test-specification language, and we use FQL in the example above to describe the condition, i.e., the already-covered test goals. FQL is a versatile language for defining various test criteria, allows to define explicit sets of test goals by enumerating single locations, but it also supports to specify full program paths as test goals, as well as value constraints on variables at certain program locations, and of course standard coverage criteria such as branch coverage are provided. For the first version of our tool implementation CondTest, we started with a simpler way to denote test goals.

Since existing testers do not accept conditions, we propose a testability transformation called `reducer` that uses the coverage criterion and the condition to transform the program under test into a *residual program* that is restricted to those parts of the program that are needed to generate test cases for the remaining test goals. This residual program is then given to an off-the-shelf tester (instead of the original program under test), such that the tester is forced to generate test cases for the remaining test goals. The resulting test suite is given to an `extractor` that extracts the test goals that are covered in the original program, and computes the new condition. This process of transforming off-the-shelf testers into conditional testers can be split into three independent components: `reducer`, `tester`, and `extractor`. All three components are defined through their type and soundness-requirements, and many different implementations are possible.

To show the potential of our approach, we implemented examples for `reducer` and `extractor`. We use the common formats and infrastructure of the International Competition on Software Testing (Test-Comp) [2] to allow plug-and-play transformation of existing software testers (for example, CoVeriTest, CPA/Tiger, Klee) into conditional testers.

In addition, we contribute a construction based on conditional testing that turns an existing, formal software verifier into a conditional tester, such that existing verifiers and existing testers can be combined as well. Formal verifiers can be specialized for finding a counterexample to a certain specification, e.g., an assertion violation or a program location of interest. Verifiers have been able to create test cases from such counterexamples for over a decade [3,39] and can thus be used for directed generation of test cases for hard-to-reach test goals. Our generic conditional tester can use all verifiers (31 tools in 2019) of the International Competition on Software Verification (SV-COMP) [1]. It uses the standard violation-witness exchange-format [4] and transforms created witnesses into executable tests [5]. To feed test goals to verifiers, we provide a tailored transformation that inserts function calls at test goals and defines the specification such that the verifier shall prove unreachability of such a function call. Since most verifiers stop their analysis after finding one counterexample (i.e., creating a single test case), we repeatedly apply conditional testing with the same verifier to obtain a full test suite.

Related Work. We base our work on conditional model checking [7], which is a general concept for information exchange between different model checkers through the use of *conditions*. The conditions are used to instruct the next conditional model checker which parts of the state space it does not need to verify because the previous model checker had successfully verified those parts of the state space already. To transform any off-the-shelf model checker into a conditional model checker, program reduction [9,18] was proposed and successfully applied. We apply this general idea to testing and call it conditional testing. The conditions of conditional testing describe parts of the program that do not need to be tested, in terms of test goals. Similar to the reducer for conditional model checking [9] (which cuts off program paths that are already verified), we developed a reducer that cuts off program paths whose test goals are already covered. Further transformation techniques that reduce programs to only contain program paths that may be relevant for analysis include program slicing [18,38] and program trimming [20].

Other works that allow combinations of different testing techniques exists; they are either limited to specific test-case-generation techniques [8,28,30,31,36,40] or require changes of the existing implementations [8,31]. In contrast, conditional testing is completely technique-agnostic and works with existing testers 'off the shelf', that is, without changing the existing testers. Some techniques of test-suite augmentation [27,37] can be used to iteratively generate test suites with one arbitrary tester, and one specific second technique that reuses the test suite generated by the first tester. These approaches are subsumed by conditional testing as special cases. Further combination approaches of tools for verification and testing include the Electronic Tools Integration platform (ETI) [29,35], and the Evidential Tool Bus (ETB) [17,32]. Conditional model checking was also applied to combine program analysis and testing [13,16,18].

Contributions. This article describes the following contributions:

1. We introduce the concept of *conditional testing* (Fig. 1), which enables quick and simple combinations of conditional testers with information passing. This provides the interface to combine existing testers.
2. We present a construction of conditional testers from *off-the-shelf test-case generators*, based on program reduction and test-goal extraction (Sect. 3).
3. We present several possible combinations for conditional testers (Sect. 4).
4. Using some of these combinations, we present a construction of testers from *off-the-shelf software verifiers*, based on conditional testing (Sect. 5).
5. We have implemented the generic conditional tester CondTest, which contains all components that are necessary for the above-mentioned constructions and combinations (`https://doi.org/10.5281/zenodo.3352401`).
6. We show the potential of conditional testing for software via a thorough experimental evaluation on the large Test-Comp benchmark set, consisting of 1 720 benchmark tasks (Sect. 6).

2 Background

In the following, we remind the reader of some notions that are necessary to instantiate the concept of conditional testing to software. A *test vector* $\bar{v} = \langle v_0, \ldots, v_n \rangle$ is a sequence of program inputs v_i with $0 \leq i \leq n$. A test vector describes a test case over the program inputs, in the order that they are passed to the program under test. A *test suite* $\{\bar{v}_0, \ldots, \bar{v}_l\}$ is a set of test vectors $\bar{v}_i$ with $0 \leq i \leq l$. We store and exchange test suites in the Test-Comp test format [2]. All Test-Comp participants can write a generated test suite in this format, which stores a test suite in a test-suite directory with several files in XML format: (1) one metadata file that contains metadata about the created test suite, and (2) one additional file for each test vector. Each test-vector file lists the test values of that test case.

We represent programs as *control-flow automata* (CFA) [6]. A CFA is an automaton $P = (L, l_0, E)$ with a set L of states, initial state l_0, and a set $E = L \times Ops \times L$ of edges, with set Ops of all possible program operations. The set L of states represents the program locations, the initial state l_0 represents the entry point of the program, and each control-flow edge $(l, op, l') \in E$ represents a program transfer where the control flows from program location l to program location l' and program operation op is executed. An operation is either an *assignment*, an *assumption*, or a `nop`. An assignment $x := exp$ assigns the value of expression exp to program variable x, where exp is a either a constant or an

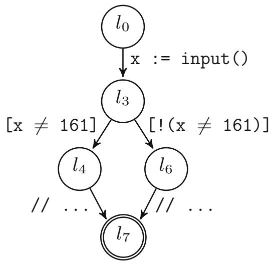


Fig. 4. CFA representation of the program in Fig. 2

arithmetic expression over constants and program variables. An assumption $[p]$ only transfers control from l to l' if p is true, where p is a boolean expression over constants and program variables. A `nop` is a program operation with no effect on the program's data state. A `nop` may have an arbitrary text label. Figure 4 shows a CFA representation of the program from our introductory example (Fig. 2). A *program path* $\pi = \langle l_0 \xrightarrow{op_0} l_1 \xrightarrow{op_1} \dots l_{n-1} \xrightarrow{op_{n-1}} l_n \rangle$ is a sequence of program locations that are sequentially connected through CFA edges $(l_i, op_i, l_{i+1}) \in E$. We write $\pi \in [\![P]\!]$ if π is a program path of program P. The execution of a test vector $\bar{v}$ on a CFA P results in a single, deterministic program path $\langle l_0 \xrightarrow{op_0} \dots \xrightarrow{op_{n-1}} l_n \rangle$, beginning at the program entry l_0. A test vector *covers* a test goal g if its execution results in a program path that reaches g.

A *violation witness* [4] is a non-deterministic, finite-state automaton that describes a set of program paths from which at least one reaches a specification violation. From each violation witness, at least one test vector can be extracted [5] that follows a program path described by the witness.

A testability transformation [23] is a transformation $\mathcal{P} \times \mathcal{G} \to \mathcal{P} \times \mathcal{G}$ over the set $\mathcal{P}$ of programs and the set $\mathcal{G}$ of test-goal descriptions. A testability transformation τ transforms a given program P and given test goals G such that, for $\tau(P, G) = (P', G')$, the following holds: if a test-suite S covers all test goals G' on P', test suite S covers all test goals G on P. The reducer presented in the following section will be based on a testability transformation that only transforms the program and keeps the test goals unchanged.

3 Construction of Conditional Testers from Existing Testers

Figure 5 shows how a conditional tester can be created from an off-the-shelf tester. A conditional software tester gets as input a program under test P, a coverage criterion φ, and a condition Ψ_0 (that describes already covered test goals). First, the set $G = [\![\varphi]\!] \setminus \Psi_0$ of remaining test goals that shall be covered is computed. Then, a program reducer `reducer` takes G and P, and creates a residual program that contains the program behavior relevant for creating test cases that cover test goals in G and that omits other program behavior. This residual program and coverage criterion φ are then given to a (classic, existing)

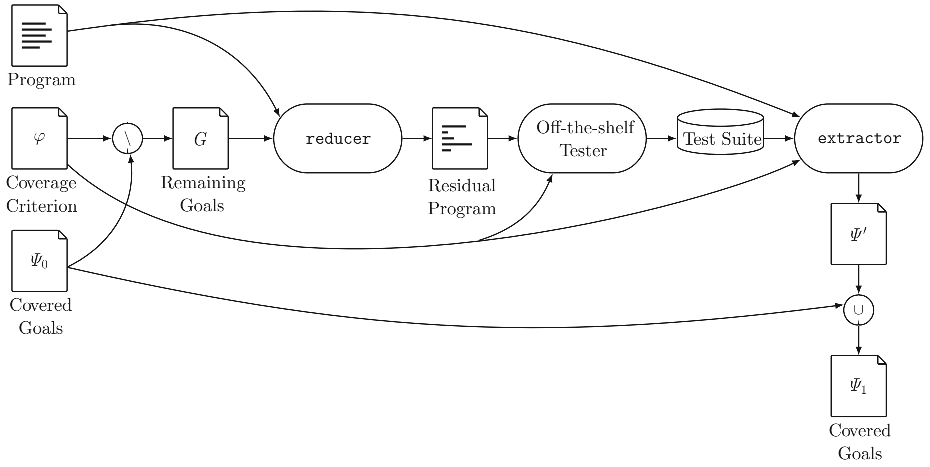


Fig. 5. Conditional tester $\texttt{tester}^{\texttt{cond}}$

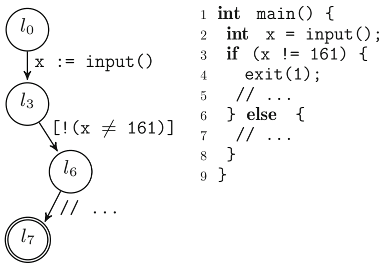
tester, which creates new test cases based on them. Once the tester stops, the original program P, the coverage criterion φ, and the created test suite are given to a test-goal extractor `extractor`, which computes all test goals Ψ' in P that are described by φ and that the test suite covers. Then, the newly covered goals Ψ' are combined with Ψ_0 to get the full set $\Psi_1 = \Psi_0 \cup \Psi'$ of now covered test goals.

In the following, we will show requirements on the components `reducer` and `extractor`. We consider programs in their CFA representation. For ease of presentation, we assume that all program variables and constants are integers, and we only consider intra-procedural analysis here, i.e., programs with a single procedure. Our approach can be naturally extended to other data types and inter-procedural analysis. We represent test goals as CFA edges and describe conditions as sets of test goals.[1]

3.1 Program Reduction

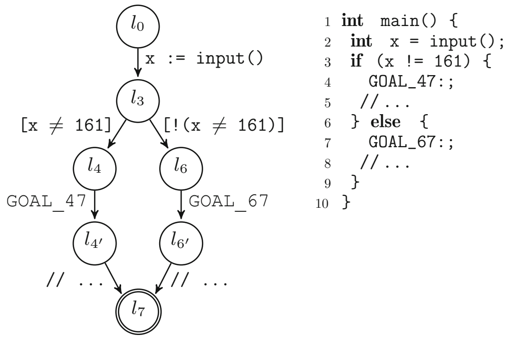
A program reducer is a testability transformation $\texttt{reducer}_G : P \to P'$ that transforms, for a given set G of test goals, a program P to a program P' that is G-coverage-equivalent to P. Two programs P and P' are *G-coverage-equivalent* if the two executions of P and P' on a test vector $\bar{v}$ cover the same subset $G_{\bar{v}} \subseteq G$ of test goals. Compared to traditional testability transformation [23], the set G of test goals is not changed by `reducer` (we write $\texttt{reducer}_G : P \to P'$ as abbreviation for $\texttt{reducer} : P \times G \to P' \times G$). This allows us to run testers and generated test cases on the same coverage criterion, and no mapping between test goals is necessary. We require a program reducer to be *sound* and *complete*. Soundness is the basic requirement for testability transformations [23]. We also require completeness to ensure that test-case generation on the reduced program does not miss any test goal that is reachable in the original program.

[1] All coverage criteria that are based on code reachability can be reduced to reachability of CFA edges through testability transformations [23,33].



```
1 int main() {
2  int x = input();
3  if (x != 161) {
4    exit(1);
5   // ...
6  } else {
7   // ...
8  }
9 }
```

Fig. 6. Residual program for test goal $(l_6, \texttt{// ...}, l_7)$



```
int main() {
 int x = input();
 if (x != 161) {
   GOAL_47:;
  //...
 } else {
   GOAL_67:;
  //...
 }
}
```

Fig. 7. Program instrumented by Alg. 1 for test-goal extraction

Soundness. Given a program P and a set G of test goals, the reducer $\texttt{reducer}_G$ is *sound* if the following holds: if a test vector $\bar{v}$ on program $P' = \texttt{reducer}_G(P)$ covers a test goal $g \in G$, then $\bar{v}$ on program P covers g.

Completeness. Given a program P and a set G of test goals, the reducer $\texttt{reducer}_G$ is *complete* if the following holds: if a test vector $\bar{v}$ on program P covers a test goal $g \in G$, then $\bar{v}$ on program $P' = \texttt{reducer}_G(P)$ covers g.

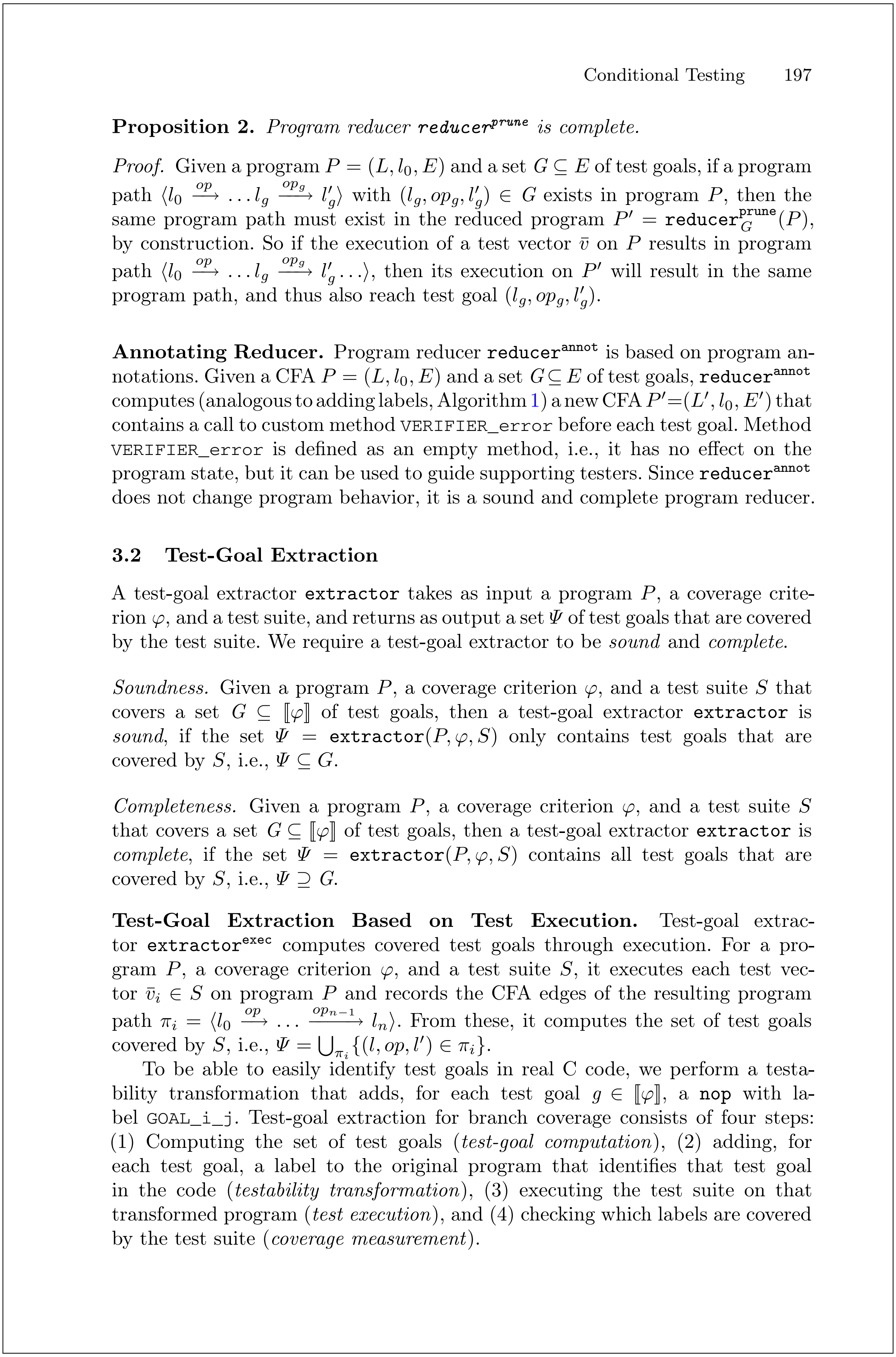
Identity Reducer. The program reducer $\texttt{reducer}^{\texttt{id}}$ is the identity, i.e., it returns a given program without any modification.

Pruning Reducer. The program reducer $\texttt{reducer}^{\texttt{prune}}$ is based on syntactic reachability. Given a CFA $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, $\texttt{reducer}^{\texttt{prune}}$ computes a new CFA $P' = (L', l_0, E')$ that only contains program locations and their corresponding edges from which a test goal is reachable. Formally, $L' = \{l \in L \mid \exists (l_g, op_g, l'_g) \in G : \langle \ldots \xrightarrow{op} l \xrightarrow{op'} \ldots l_g \xrightarrow{op_g} l'_g \rangle \in [\![P]\!]\}$ and $E' = \{(l, op, l') \in E \mid l, l' \in L'\}$.

Figure 6 shows the result of $\texttt{reducer}^{\texttt{prune}}_{\{(l_6, \texttt{// ...}, l_7)\}}(P)$ for our example program (Fig. 4), as CFA and translated to C code. Because the left branch with condition $x \neq 161$ can never reach test goal $(l_6, \texttt{// ...}, l_7)$, it is removed from the CFA. C code can not express single assumption edges, so we translate the CFA by placing an `exit`-call after the first assumption that is not part of the CFA (line 4).

Proposition 1. *Program reducer* $\texttt{reducer}^{\texttt{prune}}$ *is sound.*

Proof. Given a program $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, if a program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \rangle$ with $(l_g, op_g, l'_g) \in G$ exists in program $P' = \texttt{reducer}^{\texttt{prune}}_G(P)$, then the same program path must exist in the original program P, by construction. So if the execution of a test vector $\bar{v}$ on P' results in program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \ldots \rangle$, then its execution on P will result in the same program path, and thus also reach test goal (l_g, op_g, l'_g).



Proposition 2. *Program reducer* $\texttt{reducer}^{prune}$ *is complete.*

Proof. Given a program $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, if a program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \rangle$ with $(l_g, op_g, l'_g) \in G$ exists in program P, then the same program path must exist in the reduced program $P' = \texttt{reducer}^{\texttt{prune}}_{G}(P)$, by construction. So if the execution of a test vector $\bar{v}$ on P results in program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \ldots \rangle$, then its execution on P' will result in the same program path, and thus also reach test goal (l_g, op_g, l'_g).

Annotating Reducer. Program reducer $\texttt{reducer}^{\texttt{annot}}$ is based on program annotations. Given a CFA $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, $\texttt{reducer}^{\texttt{annot}}$ computes (analogous to adding labels, Algorithm 1) a new CFA $P' = (L', l_0, E')$ that contains a call to custom method `VERIFIER_error` before each test goal. Method `VERIFIER_error` is defined as an empty method, i.e., it has no effect on the program state, but it can be used to guide supporting testers. Since $\texttt{reducer}^{\texttt{annot}}$ does not change program behavior, it is a sound and complete program reducer.

3.2 Test-Goal Extraction

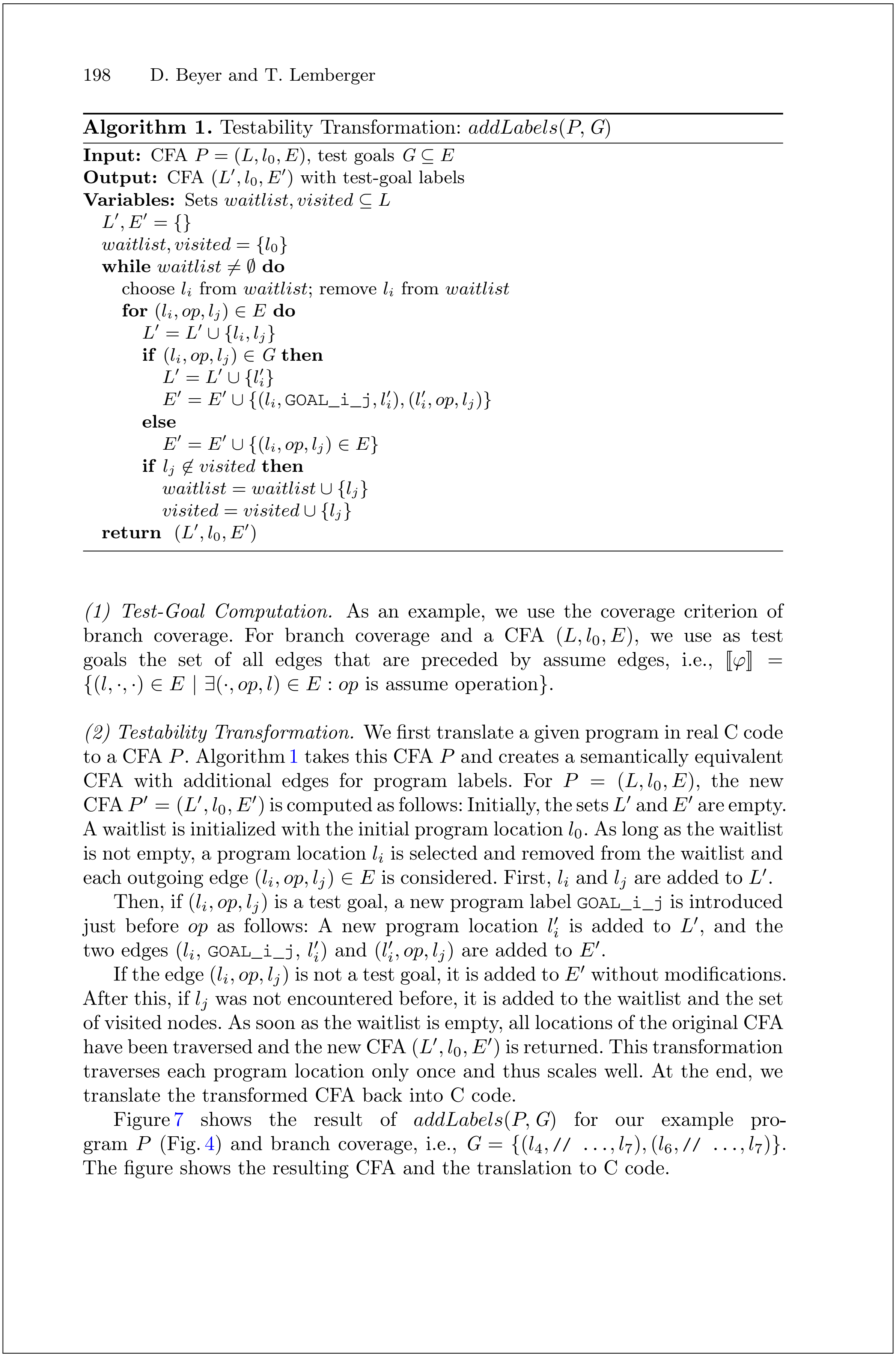
A test-goal extractor `extractor` takes as input a program P, a coverage criterion φ, and a test suite, and returns as output a set Ψ of test goals that are covered by the test suite. We require a test-goal extractor to be *sound* and *complete*.

Soundness. Given a program P, a coverage criterion φ, and a test suite S that covers a set $G \subseteq [\![\varphi]\!]$ of test goals, then a test-goal extractor `extractor` is *sound*, if the set $\Psi = \texttt{extractor}(P, \varphi, S)$ only contains test goals that are covered by S, i.e., $\Psi \subseteq G$.

Completeness. Given a program P, a coverage criterion φ, and a test suite S that covers a set $G \subseteq [\![\varphi]\!]$ of test goals, then a test-goal extractor `extractor` is *complete*, if the set $\Psi = \texttt{extractor}(P, \varphi, S)$ contains all test goals that are covered by S, i.e., $\Psi \supseteq G$.

Test-Goal Extraction Based on Test Execution. Test-goal extractor $\texttt{extractor}^{\texttt{exec}}$ computes covered test goals through execution. For a program P, a coverage criterion φ, and a test suite S, it executes each test vector $\bar{v}_i \in S$ on program P and records the CFA edges of the resulting program path $\pi_i = \langle l_0 \xrightarrow{op} \ldots \xrightarrow{op_{n-1}} l_n \rangle$. From these, it computes the set of test goals covered by S, i.e., $\Psi = \bigcup_{\pi_i} \{(l, op, l') \in \pi_i\}$.

To be able to easily identify test goals in real C code, we perform a testability transformation that adds, for each test goal $g \in [\![\varphi]\!]$, a `nop` with label `GOAL_i_j`. Test-goal extraction for branch coverage consists of four steps: (1) Computing the set of test goals (*test-goal computation*), (2) adding, for each test goal, a label to the original program that identifies that test goal in the code (*testability transformation*), (3) executing the test suite on that transformed program (*test execution*), and (4) checking which labels are covered by the test suite (*coverage measurement*).



Algorithm 1. Testability Transformation: *addLabels*(P, G)

```
Input: CFA $P = (L, l_0, E)$, test goals $G \subseteq E$
Output: CFA $(L', l_0, E')$ with test-goal labels
Variables: Sets $waitlist, visited \subseteq L$
  $L', E' = \{\}$
  $waitlist, visited = \{l_0\}$
  while $waitlist \neq \emptyset$ do
    choose $l_i$ from $waitlist$; remove $l_i$ from $waitlist$
    for $(l_i, op, l_j) \in E$ do
      $L' = L' \cup \{l_i, l_j\}$
      if $(l_i, op, l_j) \in G$ then
        $L' = L' \cup \{l'_i\}$
        $E' = E' \cup \{(l_i, \texttt{GOAL\_i\_j}, l'_i), (l'_i, op, l_j)\}$
      else
        $E' = E' \cup \{(l_i, op, l_j) \in E\}$
      if $l_j \notin visited$ then
        $waitlist = waitlist \cup \{l_j\}$
        $visited = visited \cup \{l_j\}$
  return $(L', l_0, E')$
```

(1) Test-Goal Computation. As an example, we use the coverage criterion of branch coverage. For branch coverage and a CFA (L, l_0, E), we use as test goals the set of all edges that are preceded by assume edges, i.e., $[\![\varphi]\!] = \{(l, \cdot, \cdot) \in E \mid \exists(\cdot, op, l) \in E : op$ is assume operation$\}$.

(2) Testability Transformation. We first translate a given program in real C code to a CFA P. Algorithm 1 takes this CFA P and creates a semantically equivalent CFA with additional edges for program labels. For $P = (L, l_0, E)$, the new CFA $P' = (L', l_0, E')$ is computed as follows: Initially, the sets L' and E' are empty. A waitlist is initialized with the initial program location l_0. As long as the waitlist is not empty, a program location l_i is selected and removed from the waitlist and each outgoing edge $(l_i, op, l_j) \in E$ is considered. First, l_i and l_j are added to L'.

Then, if (l_i, op, l_j) is a test goal, a new program label `GOAL_i_j` is introduced just before *op* as follows: A new program location l'_i is added to L', and the two edges $(l_i$, `GOAL_i_j`, $l'_i)$ and (l'_i, op, l_j) are added to E'.

If the edge (l_i, op, l_j) is not a test goal, it is added to E' without modifications. After this, if l_j was not encountered before, it is added to the waitlist and the set of visited nodes. As soon as the waitlist is empty, all locations of the original CFA have been traversed and the new CFA (L', l_0, E') is returned. This transformation traverses each program location only once and thus scales well. At the end, we translate the transformed CFA back into C code.

Figure 7 shows the result of *addLabels*(P, G) for our example program P (Fig. 4) and branch coverage, i.e., $G = \{(l_4, \texttt{// \ldots}, l_7), (l_6, \texttt{// \ldots}, l_7)\}$. The figure shows the resulting CFA and the translation to C code.

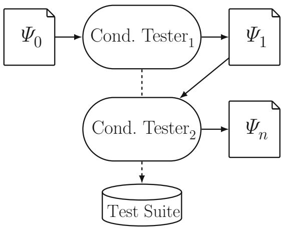


Fig. 8. `tester`$^{\texttt{seq}}$

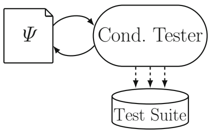


Fig. 9. `tester`$^{\texttt{cyc}}$

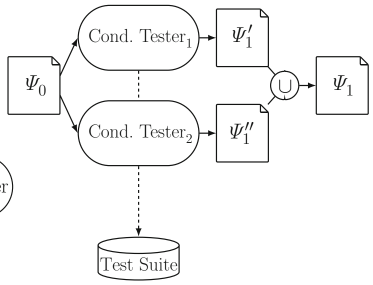


Fig. 10. `tester`$^{\texttt{par}}$

(3) Test Execution. We execute all test cases of the given test suite on the transformed program as follows: We generate a test harness that reads test values from the standard input and provides the test values to the C program. We compile this test harness with the transformed program and feed each test vector to the harness in individual executions.

(4) Coverage Measurement. We use GCov to obtain a coverage report that lists for each line[2] of the transformed C program whether it was covered by the test suite. From this report, `extractor`$^{\texttt{exec}}$ extracts the program labels of test goals that are covered, and returns the corresponding test goals.

Since `extractor`$^{\texttt{exec}}$ is based on concrete execution of the test suite on a semantically equivalent program, the method is assumed to be both sound and complete.

4 Combinations of Conditional Testers

Conditional testing enables versatile combinations of testers. We have already seen a sequential combination in the introduction (Fig. 3), but it is also possible to combine conditional testers in other ways, such as in cycles, in general portfolios (i.e., also parallel), with strategy selection, or for compositional reasoning. In the following, we will present different possible combinations of conditional testers to show the potential of conditional software testing. Note that all of these combinations are themselves conditional testers, so they can be combined with each other in any way. From now on, we will omit the program under test and the coverage criterion in figures, to have simpler diagrams.

Sequential Tester. A sequential tester `tester`$^{\texttt{seq}}(T_1, T_2)$ (Fig. 8) consists of two component testers T_1 and T_2 that are executed sequentially to generate test cases. Several sequential testers can be used to sequentially combine an arbitrary number of testers. For simplicity, we write `tester`$^{\texttt{seq}}(T_1, T_2, T_3)$ for `tester`$^{\texttt{seq}}(T_1, $`tester`$^{\texttt{seq}}(T_2, T_3))$. Each tester provides the covered test goals

[2] Since the transformed program is generated such that each operation is written on an own line in the output code, a line uniquely identifies a test-goal label.

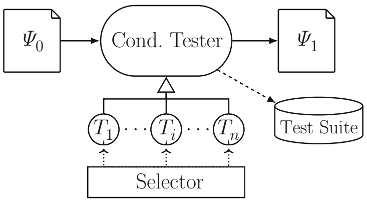


Fig. 11. $\texttt{tester}^{\texttt{select}}$

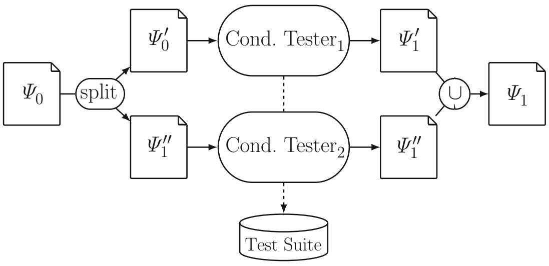


Fig. 12. $\texttt{tester}^{\texttt{comp}}$

after its run, and the set of remaining test goals will decrease. This can be used to combine strengths of different testers without further knowledge about them; testers can either get a certain time limit each, or stop early if they encounter a program feature they don't support.

Cyclic Tester. A cyclic tester $\texttt{tester}^{\texttt{cyc}}(T)$ (Fig. 9) iteratively calls a conditional tester T with the increasing set Ψ of covered test goals. This can be used, for example, to restart a tester after a certain limit is reached (e.g., memory consumption or size of path constraints in symbolic execution). In combination with $\texttt{tester}^{\texttt{seq}}$, this can also be used to cycle through a sequence of testers (round-robin principle).

Parallel Tester. A parallel tester $\texttt{tester}^{\texttt{par}}(T_1, T_2)$ (Fig. 10) runs testers T_1 and T_2 in parallel on the same inputs. Each tester produces its own set Ψ'_1, Ψ''_1 of covered test goals, and their union $\Psi'_1 \cup \Psi''_1 = \Psi_1$ is the final set of covered test goals. Several parallel testers can be used to combine an arbitrary number of testers, similar to $\texttt{tester}^{\texttt{seq}}$. In contrast to $\texttt{tester}^{\texttt{seq}}$, there is no information exchange between testers T_1 and T_2, so they may do redundant work.

Strategy-Selection Tester. A strategy-selection tester $\texttt{tester}^{\texttt{select}}(T_1, \ldots, T_n)$ (Fig. 11) uses a selector function to select to which of testers $T_1, \ldots, T_n$ the task of test-case generation is delegated. The selector function can be an arbitrary function that returns one of T_1 to T_n, e.g., a random selection, or based on a selection model that selects the most suited tester based on features of the program under test.

Compositional Tester. A compositional tester $\texttt{tester}^{\texttt{comp}}(T_1, T_2)$ (Fig. 12) first splits the condition Ψ_0 into two sets Ψ'_0 and Ψ''_0, so that $\Psi_0 = \Psi'_0 \cup \Psi''_0$. Then, tester T_1 gets as input the first set Ψ'_0, and tester T_2 gets as input the second set Ψ''_0. Both testers work on the original program P and original coverage criterion φ, but due to Ψ'_0 and Ψ''_0, the first tester only works on test goals $[\![\varphi]\!] \setminus \Psi'_0$, and the second tester only works on $[\![\varphi]\!] \setminus \Psi''_0$. They produce individual sets Ψ'_1 and Ψ''_1 of covered test goals. These are then merged into the final set $\Psi_1 = \Psi'_1 \cup \Psi''_1$ of now covered test goals. More than two testers can be combined compositionally through nested combinations. With $\texttt{tester}^{\texttt{comp}}$, work can be split (decomposition principle), for example for parallelization or to let each tester solve the test goals it is most suited for.

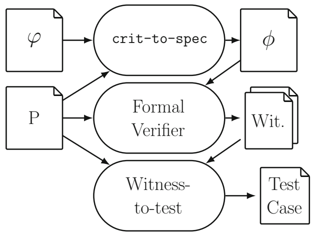


Fig. 13. $\texttt{tester}^{\texttt{veri}}$

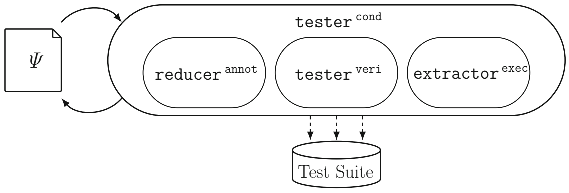


Fig. 14. $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$

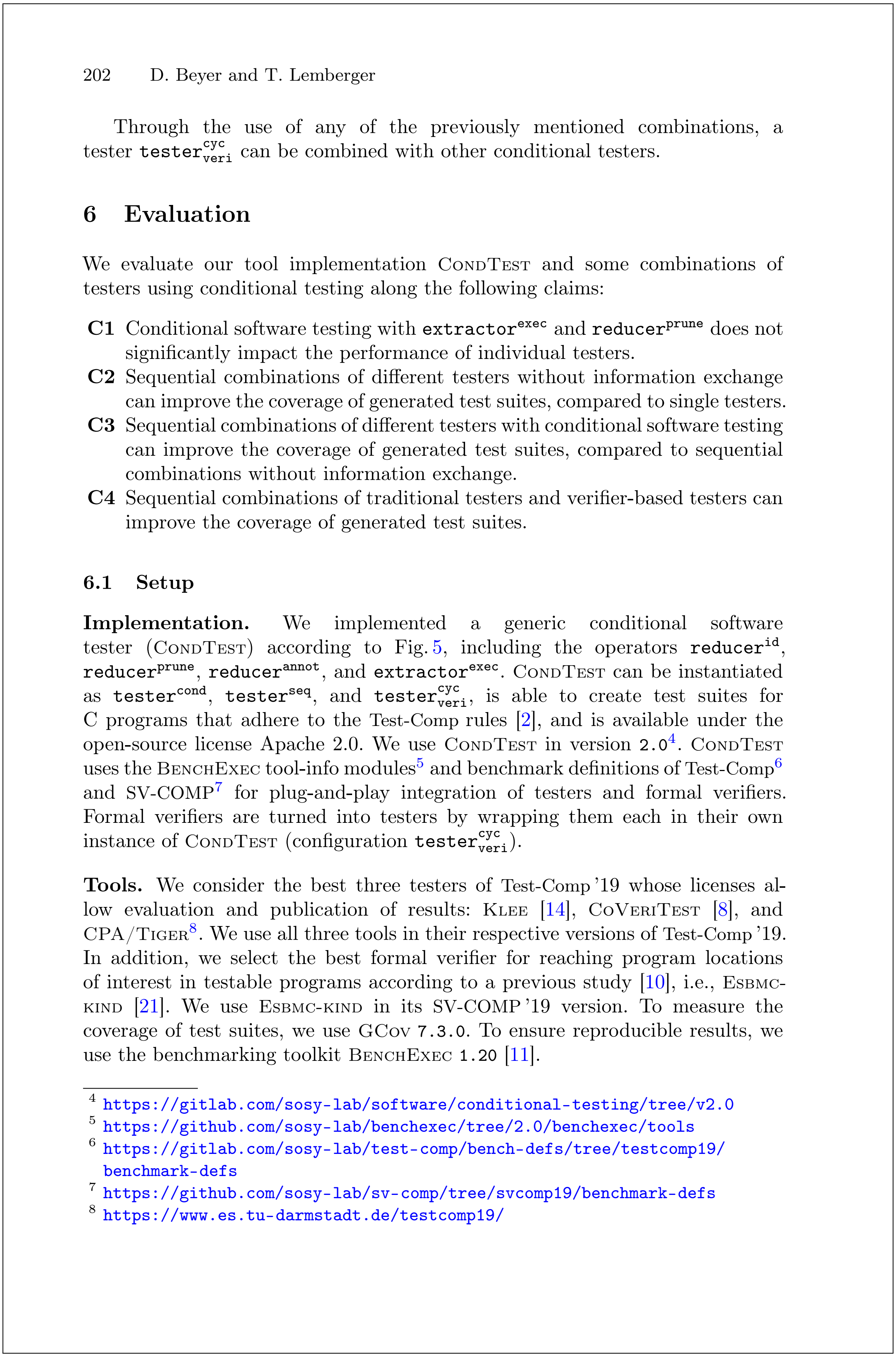
5 Construction of Conditional Testers from Existing Verifiers

It has long been possible to use formal verification of reachability properties to generate tests [3]. Compared to testers, many formal verification techniques specialize on finding single program paths to specific program states or program locations of interest; this makes them suitable for hard-to-reach test goals [10]. Figure 13 shows the (non-conditional) tester $\texttt{tester}^{\texttt{veri}}(V)$ that is based on a formal verifier V: First, function `crit-to-spec` transforms the coverage criterion φ, based on program P, to a safety specification ϕ which is constructed such that P violates ϕ if P covers a test goal from $[\![\varphi]\!]$. Then, ϕ and P are given to formal verifier V, which checks whether P satisfies ϕ. The verifier outputs one or more violation witnesses if test goals are reachable. From these violation witnesses, test cases are created by witness-to-test [5].

We use the established formats for input programs and specifications for the reachability category of SV-COMP[3] to get access to a large catalog of tools for formal verification. There are two adaptations necessary for using SV-COMP verifiers: (1) they are only required to support the property "no call to method `__VERIFIER_error` is reachable", and may not support more general reachability properties, and (2) they are only required to output a single violation witness, and thus will always lead to a test suite that only consists of one test case.

We solve both issues in the following way: We let `crit-to-spec` always return the specification that no call to `__VERIFIER_error` is reachable. We then take $\texttt{tester}^{\texttt{veri}}$ and construct from it a conditional tester based on $\texttt{tester}^{\texttt{cond}}$ with program reducer $\texttt{reducer}^{\texttt{annot}}$ and test-goal extractor $\texttt{extractor}^{\texttt{exec}}$. At this point, we have a conditional tester that uses a formal verifier to always produce a test suite with a single test case, and that returns the set of test goals covered by that test case. To produce a full test suite for all test goals, we use a cyclic tester $\texttt{tester}^{\texttt{cyc}}(\texttt{tester}^{\texttt{cond}}(\texttt{tester}^{\texttt{veri}}(V)))$ (Fig. 14). After each test-case generation run, the newly created test case is used by $\texttt{extractor}^{\texttt{exec}}$ to update the covered test goals. Then, $\texttt{reducer}^{\texttt{annot}}$ will insert calls to `__VERIFIER_error` for the remaining test goals, and $\texttt{tester}^{\texttt{veri}}(V)$ will create a new test case that covers at least one of the remaining test goals. We use $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}(V)$ to denote a verifier-based tester that is constructed from formal verifier V.

[3] https://sv-comp.sosy-lab.org/2019/rules.php



Through the use of any of the previously mentioned combinations, a tester $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$ can be combined with other conditional testers.

6 Evaluation

We evaluate our tool implementation CondTest and some combinations of testers using conditional testing along the following claims:

C1 Conditional software testing with $\texttt{extractor}^{\texttt{exec}}$ and $\texttt{reducer}^{\texttt{prune}}$ does not significantly impact the performance of individual testers.
C2 Sequential combinations of different testers without information exchange can improve the coverage of generated test suites, compared to single testers.
C3 Sequential combinations of different testers with conditional software testing can improve the coverage of generated test suites, compared to sequential combinations without information exchange.
C4 Sequential combinations of traditional testers and verifier-based testers can improve the coverage of generated test suites.

6.1 Setup

Implementation. We implemented a generic conditional software tester (CondTest) according to Fig. 5, including the operators $\texttt{reducer}^{\texttt{id}}$, $\texttt{reducer}^{\texttt{prune}}$, $\texttt{reducer}^{\texttt{annot}}$, and $\texttt{extractor}^{\texttt{exec}}$. CondTest can be instantiated as $\texttt{tester}^{\texttt{cond}}$, $\texttt{tester}^{\texttt{seq}}$, and $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$, is able to create test suites for C programs that adhere to the Test-Comp rules [2], and is available under the open-source license Apache 2.0. We use CondTest in version `2.0`[4]. CondTest uses the BenchExec tool-info modules[5] and benchmark definitions of Test-Comp[6] and SV-COMP[7] for plug-and-play integration of testers and formal verifiers. Formal verifiers are turned into testers by wrapping them each in their own instance of CondTest (configuration $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$).

Tools. We consider the best three testers of Test-Comp '19 whose licenses allow evaluation and publication of results: Klee [14], CoVeriTest [8], and CPA/Tiger[8]. We use all three tools in their respective versions of Test-Comp '19. In addition, we select the best formal verifier for reaching program locations of interest in testable programs according to a previous study [10], i.e., Esbmc-kind [21]. We use Esbmc-kind in its SV-COMP '19 version. To measure the coverage of test suites, we use GCov `7.3.0`. To ensure reproducible results, we use the benchmarking toolkit BenchExec `1.20` [11].

[4] https://gitlab.com/sosy-lab/software/conditional-testing/tree/v2.0
[5] https://github.com/sosy-lab/benchexec/tree/2.0/benchexec/tools
[6] https://gitlab.com/sosy-lab/test-comp/bench-defs/tree/testcomp19/benchmark-defs
[7] https://github.com/sosy-lab/sv-comp/tree/svcomp19/benchmark-defs
[8] https://www.es.tu-darmstadt.de/testcomp19/

Environment. We perform our experiments on a cluster of 168 machines, each with 33 GB of memory and an Intel Xeon E3-1230 v5 CPU, with 3.4 GHz and 8 processing units (with hyper-threading). We use Ubuntu 18.04 with Linux kernel 4.4 as operating system. We limit each benchmark run to 4 processing units and a time limit of 900 s. Each run of CondTest is limited to 15.5 GB. Each individual test-case generation run (e.g., execution of CPA/Tiger) is limited to 15 GB, both for native execution and as part of CondTest. This way, each test-case generation run has the same amount of memory for both native execution and execution within CondTest. Extractor $\texttt{extractor}^{\texttt{exec}}$ uses a time limit of 3 s for each test execution, to prevent hangups in case of incomplete or non-terminating tests. To measure the achieved coverage of the complete final test suites, we execute test cases with a memory limit of 7 GB, 2 processing units, and a time limit of 3 h for each generated test suite. At this time limit, no timeouts occurred during coverage measurement.

Reproducibility and Benchmark Tasks. We use all 1 720 test tasks of the *Cover-Branches* category of the Test-Comp '19 benchmark. All of our experimental data are available online[9] and through a replication package.[10]

6.2 Results

C1: No Significant Overhead in CondTest. Figure 15 shows the branch coverage per task achieved by the test suites created by CoVeriTest, CPA/Tiger, and Klee, respectively, in their original Test-Comp configurations (x-axis), and as conditional testers $\texttt{tester}^{\texttt{cond}}$ (y-axis) inside CondTest with $\texttt{reducer}^{\texttt{prune}}$ and $\texttt{extractor}^{\texttt{exec}}$ ($\texttt{reducer}^{\texttt{prune}}$ does not really prune anything because of the full set of test goals, but parses the program, runs the pruning algorithm, and writes out the transformed C program; the idea is to find out whether this process is efficient and does not negatively impact the overall process). The CPU-time limit for each test-case generation was set to 900 s (the CPU time consumed by $\texttt{reducer}^{\texttt{prune}}$ is included in the measured CPU time, and thus, implicitly subtracted from the CPU-time available for the tester). Since $\texttt{extractor}^{\texttt{exec}}$ only runs after the tester, it has no influence on the time limit in a configuration with a single tester. For points on the diagonal, the same coverage was achieved by the original tester and its integration in $\texttt{tester}^{\texttt{cond}}$; points above the diagonal represent tasks for which $\texttt{tester}^{\texttt{cond}}$ achieved a higher coverage, and the points below the diagonal represent tasks for which $\texttt{tester}^{\texttt{cond}}$ achieved a lower coverage. For CoVeriTest, the coverage for a few tasks are a bit worse when run with CondTest (just below the diagonal). This is because CondTest uses a different, more strict technique to enforce the memory limit than the benchmarking tool BenchExec, due to technical reasons. For CPA/Tiger, outliers on the left (vertical stack of points) are due to crashes from memory exhaustion. CPA/Tiger operates close to the memory limit for many tasks. Because of this, small variations in

[9] https://www.sosy-lab.org/research/conditional-testing/
[10] https://doi.org/10.5281/zenodo.3352401

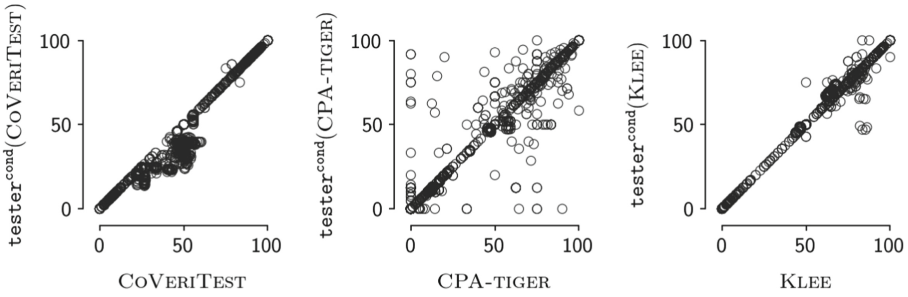


Fig. 15. Branch coverage of test suites created by original tools vs. their integration in $\texttt{tester}^{\texttt{cond}}$ (in percent)

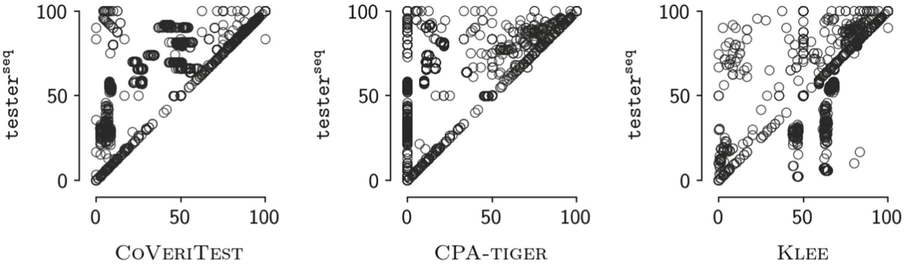


Fig. 16. Branch coverage of test suites created by original tools vs. their sequential combinations with $\texttt{reducer}^{\texttt{id}}$, i.e., without information exchange (in percent)

memory usage can lead to crashes. In our experiments, for most of these tasks it was random whether CPA/Tiger stayed closely below the memory limit, or exceeded it and crashed. Thus, the issue is not related to CondTest, but results from memory exhaustion of the native tester. Besides these issues, it is visible that for all three testers, no significant differences in branch coverage exist. This suggests that using $\texttt{tester}^{\texttt{cond}}$ with the proposed operators $\texttt{reducer}^{\texttt{prune}}$ and $\texttt{extractor}^{\texttt{exec}}$ does not lead to a significant negative impact on the performance just by using the conditional-testing construction.

C2: Combinations Can Improve Coverage. Figure 16 shows the branch coverage per task achieved by the test suites created by CoVeriTest, CPA/Tiger, and Klee, respectively, in their original Test-Comp configurations with 900 s CPU-time limit (x-axis), and the coverage per task achieved by the test suites created by CondTest (y-axis) with the sequential combination $\texttt{tester}^{\texttt{seq}}(\texttt{tester}^{\texttt{cond}}(\text{CPA/Tiger})_{300}$, $\texttt{tester}^{\texttt{cond}}(\text{CoVeriTest})_{300}$, $\texttt{tester}^{\texttt{cond}}(\text{Klee})_{300})$ and the reducer $\texttt{reducer}^{\texttt{id}}$, i.e., without information exchange between the three testers. Each single conditional tester (i.e., $\texttt{tester}^{\texttt{cond}}$ based on CPA/Tiger, CoVeriTest, and Klee) was stopped after 300 s each, and each full test-case generation $\texttt{tester}^{\texttt{seq}}$ run was stopped after a total of 900 s (the CPU time consumed by CondTest for, e.g., process management, is included in the measured CPU time, and thus, implicitly subtracted from the CPU-time available for the last tester, $\texttt{tester}^{\texttt{cond}}$(Klee)).

Table 1. Coverage of test suites generated without information reuse ($\texttt{reducer}^{\texttt{id}}$) and with information reuse through $\texttt{reducer}^{\texttt{prune}}$

Task	branch coverage		
	id	→	prune
mod3.c.v+sep-reducer	75.0	+ 5.00	80.0
Problem07_label35	52.0	+ 2.00	54.0
Problem07_label37	54.2	+ 1.97	56.2
Problem04_label35	79.5	+ 1.79	81.3
Problem06_label02	57.0	+ 1.70	58.7
Problem06_label27	57.5	+ 1.09	58.6
Problem04_label02	80.2	+ 1.06	81.3
Problem06_label18	57.5	+ 1.05	58.6
Problem04_label16	79.1	+ 1.01	80.1
Problem04_label34	80.2	+ 0.99	81.2

Table 2. Coverage of test suites generated without (`prune`) and with (`vb`) support of Esbmc-kind

Task	branch coverage		
	prune	→	vb
Problem08_label30	5.72	+ 56.2	62.0
Problem08_label32	5.72	+ 56.1	61.9
Problem08_label06	5.72	+ 56.1	61.8
Problem08_label35	5.72	+ 56.0	61.7
Problem08_label00	5.72	+ 55.9	61.6
Problem08_label11	5.72	+ 55.8	61.5
Problem08_label19	5.72	+ 55.7	61.5
Problem08_label29	5.67	+ 55.7	61.4
Problem08_label22	5.72	+ 55.7	61.5
Problem08_label56	5.72	+ 55.7	61.5

The scatter plots in Fig. 16 show that the branch coverage of the test suites created by the sequential combination is significantly higher for a significant amount of benchmark tasks. This shows that the used testers (with CPU time limit of 300 s) can complement each other well, and that combinations can perform better than a single tester running for a longer time on its own (900 s CPU time limit).

C3: Condition Passing Can Further Improve Coverage. To show that conditional software testing can lead to generated test suites with improved coverage, we compare the branch coverage of the test suites generated by CondTest with $\texttt{tester}^{\texttt{seq}}(\texttt{tester}^{\texttt{cond}}(\text{CPA/Tiger})_{300}, \texttt{tester}^{\texttt{cond}}(\text{CoVeriTest})_{300}, \texttt{tester}^{\texttt{cond}}(\text{Klee})_{300})$, and the two reducers $\texttt{reducer}^{\texttt{id}}$, i.e., without information exchange, and $\texttt{reducer}^{\texttt{prune}}$, i.e., with program reduction based on syntactic reachability. Table 1 shows a comparison of the branch coverage of test suites generated by both techniques on a selection of benchmark tasks (programs with complicated branching), rounded to three digits. It shows that information exchange can lead to generated test suites with improved branch coverage, adding up to 5 % branch coverage.

C4: Verifiers as Test-Generators Can Improve Coverage. To show that verifier-based testers can generate test suites with improved coverage compared to combinations of traditional testers, we compare the branch coverage of the test suites generated by CondTest with $\texttt{tester}^{\texttt{seq}}(\texttt{tester}^{\texttt{cond}}(\text{CPA/Tiger})_{300}, \texttt{tester}^{\texttt{cond}}(\text{CoVeriTest})_{300}, \texttt{tester}^{\texttt{cond}}(\text{Klee})_{300})$ (called `prune`) and the test suites generated by CondTest with $\texttt{tester}^{\texttt{seq}}(\texttt{tester}^{\texttt{cond}}(\text{CPA/Tiger})_{200}, \texttt{tester}^{\texttt{cond}}(\text{CoVeriTest})_{200}, \texttt{tester}^{\texttt{cond}}(\text{Klee})_{200}, \texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}(\text{Esbmc})_{300})$ (called `vb`). Both `prune` and `vb` use $\texttt{reducer}^{\texttt{prune}}$ and $\texttt{extractor}^{\texttt{exec}}$. For `prune`, each individual tester is stopped after 300 s. For `vb`, CPA/Tiger, CoVeriTest, and Klee are each stopped after 200 s, and Esbmc runs for 300 s. The total time of each run of CondTest is 900 s (i.e., the CPU time required by $\texttt{reducer}^{\texttt{prune}}$ and $\texttt{extractor}^{\texttt{exec}}$ is implicitly subtracted from the CPU time available for the last tester, i.e., Klee in `prune` and Esbmc in `vb`).

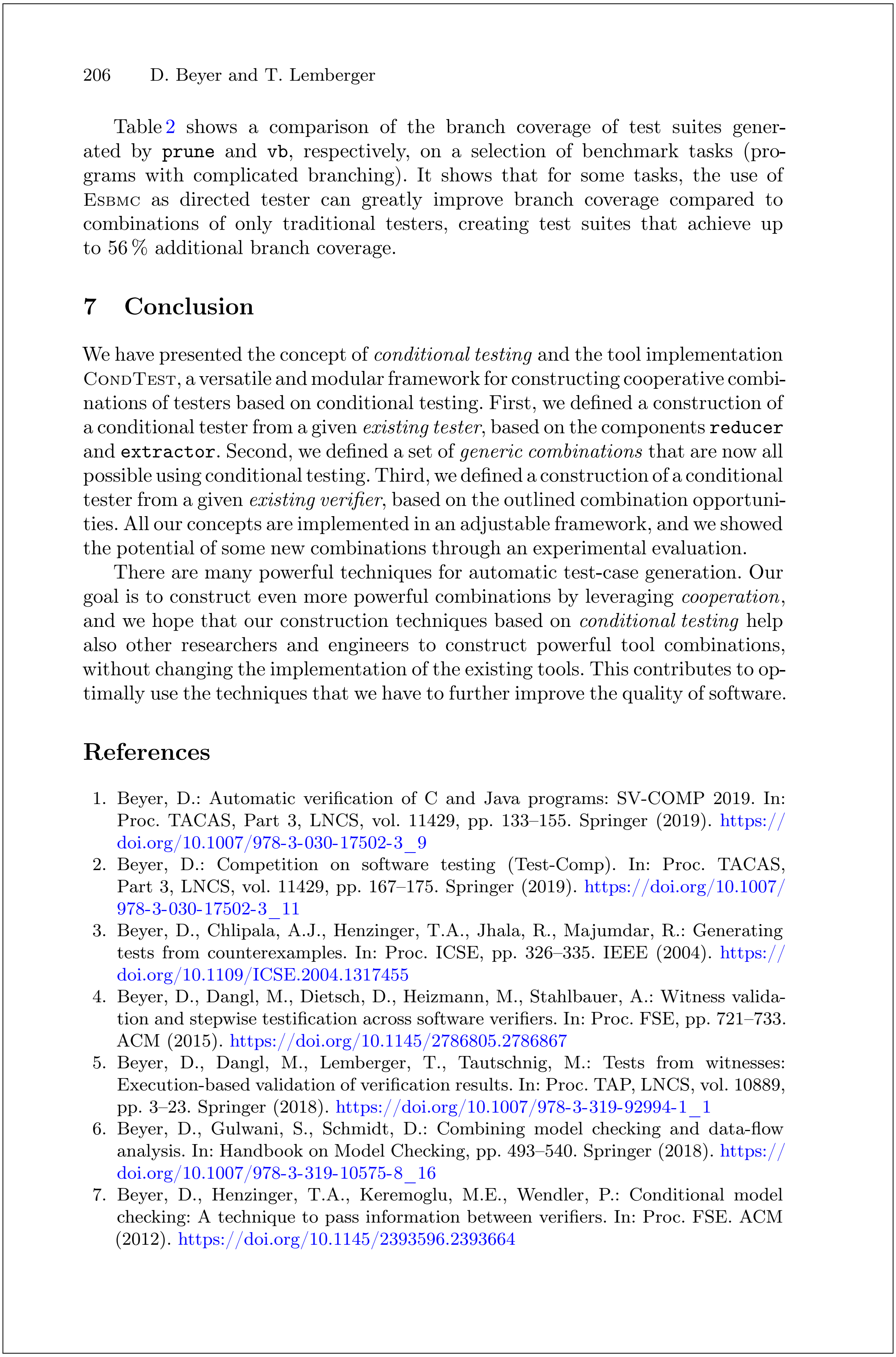


Table 2 shows a comparison of the branch coverage of test suites generated by `prune` and `vb`, respectively, on a selection of benchmark tasks (programs with complicated branching). It shows that for some tasks, the use of Esbmc as directed tester can greatly improve branch coverage compared to combinations of only traditional testers, creating test suites that achieve up to 56 % additional branch coverage.

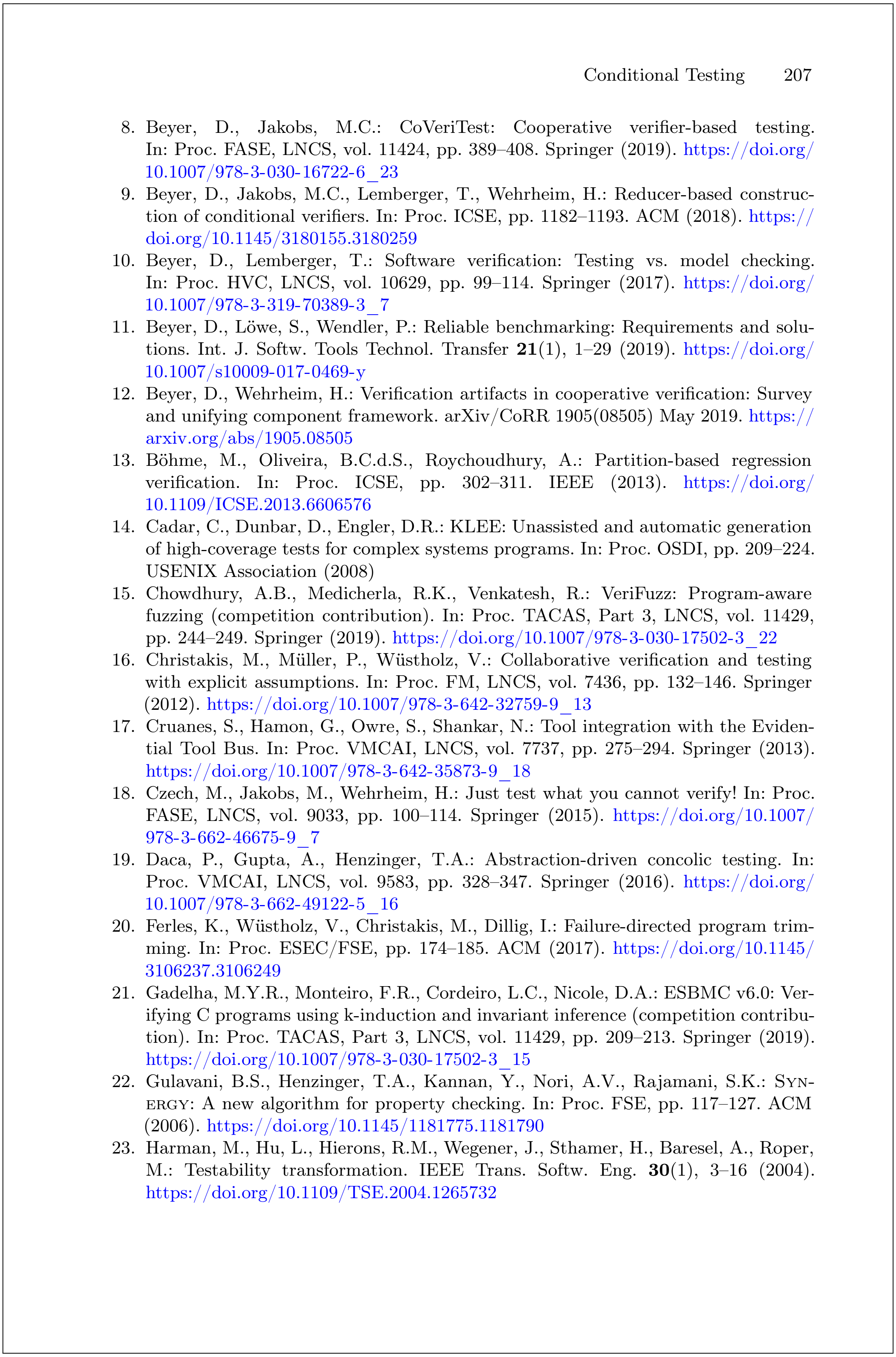
7 Conclusion

We have presented the concept of *conditional testing* and the tool implementation CondTest, a versatile and modular framework for constructing cooperative combinations of testers based on conditional testing. First, we defined a construction of a conditional tester from a given *existing tester*, based on the components `reducer` and `extractor`. Second, we defined a set of *generic combinations* that are now all possible using conditional testing. Third, we defined a construction of a conditional tester from a given *existing verifier*, based on the outlined combination opportunities. All our concepts are implemented in an adjustable framework, and we showed the potential of some new combinations through an experimental evaluation.

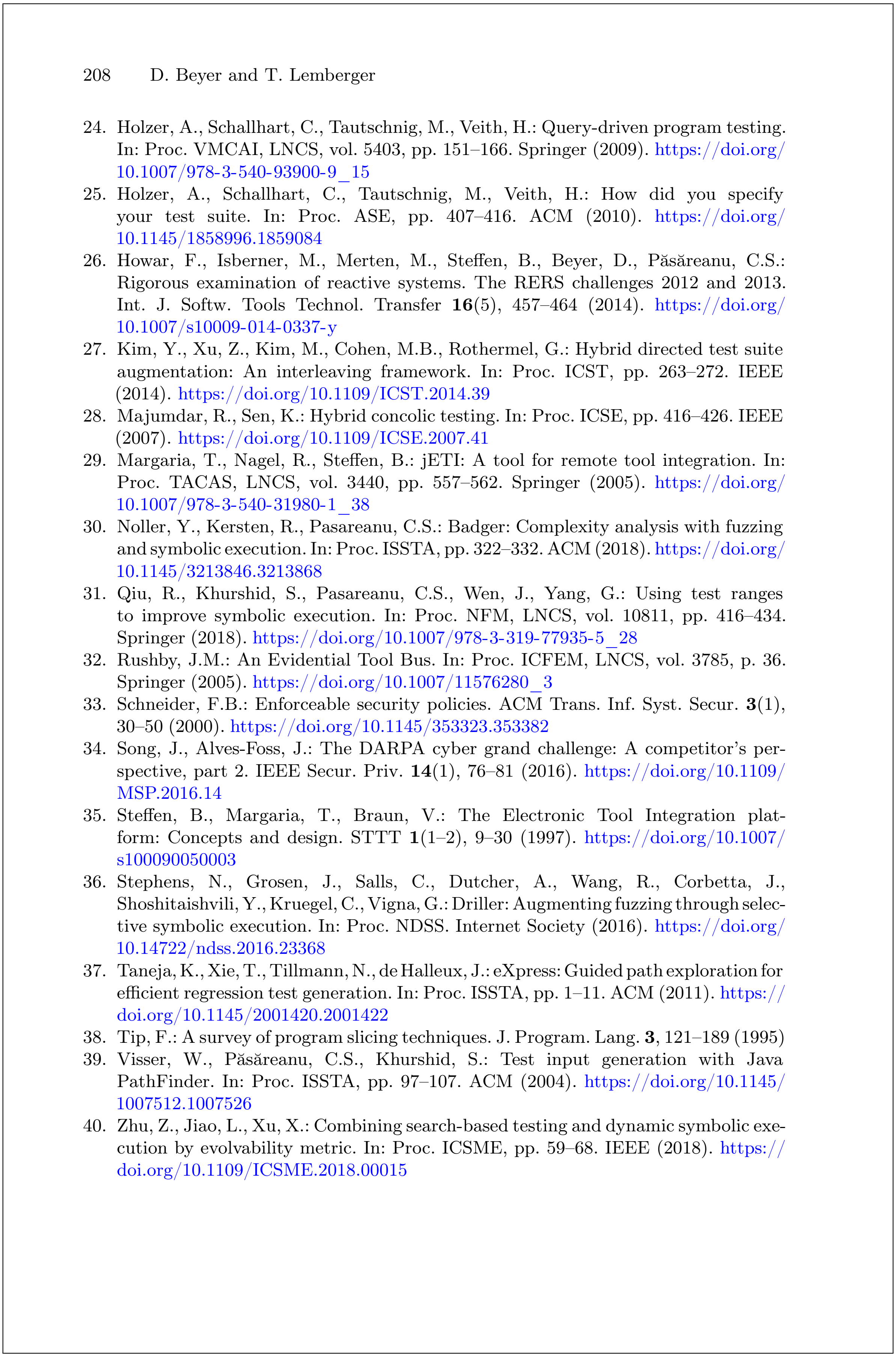
There are many powerful techniques for automatic test-case generation. Our goal is to construct even more powerful combinations by leveraging *cooperation*, and we hope that our construction techniques based on *conditional testing* help also other researchers and engineers to construct powerful tool combinations, without changing the implementation of the existing tools. This contributes to optimally use the techniques that we have to further improve the quality of software.

References

1. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 133–155. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
2. Beyer, D.: Competition on software testing (Test-Comp). In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 167–175. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_11
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE, pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
4. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE, pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
5. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP, LNCS, vol. 10889, pp. 3–23. Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
6. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook on Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
7. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664



8. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE, LNCS, vol. 11424, pp. 389–408. Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
9. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE, pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
10. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC, LNCS, vol. 10629, pp. 99–114. Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_7
11. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
12. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. arXiv/CoRR 1905(08505) May 2019. https://arxiv.org/abs/1905.08505
13. Böhme, M., Oliveira, B.C.d.S., Roychoudhury, A.: Partition-based regression verification. In: Proc. ICSE, pp. 302–311. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606576
14. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI, pp. 209–224. USENIX Association (2008)
15. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program-aware fuzzing (competition contribution). In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 244–249. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
16. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM, LNCS, vol. 7436, pp. 132–146. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13
17. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: Proc. VMCAI, LNCS, vol. 7737, pp. 275–294. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_18
18. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE, LNCS, vol. 9033, pp. 100–114. Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7
19. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Proc. VMCAI, LNCS, vol. 9583, pp. 328–347. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_16
20. Ferles, K., Wüstholz, V., Christakis, M., Dillig, I.: Failure-directed program trimming. In: Proc. ESEC/FSE, pp. 174–185. ACM (2017). https://doi.org/10.1145/3106237.3106249
21. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 209–213. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15
22. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Proc. FSE, pp. 117–127. ACM (2006). https://doi.org/10.1145/1181775.1181790
23. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Softw. Eng. **30**(1), 3–16 (2004). https://doi.org/10.1109/TSE.2004.1265732



24. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Proc. VMCAI, LNCS, vol. 5403, pp. 151–166. Springer (2009). https://doi.org/10.1007/978-3-540-93900-9_15
25. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proc. ASE, pp. 407–416. ACM (2010). https://doi.org/10.1145/1858996.1859084
26. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.S.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. Int. J. Softw. Tools Technol. Transfer **16**(5), 457–464 (2014). https://doi.org/10.1007/s10009-014-0337-y
27. Kim, Y., Xu, Z., Kim, M., Cohen, M.B., Rothermel, G.: Hybrid directed test suite augmentation: An interleaving framework. In: Proc. ICST, pp. 263–272. IEEE (2014). https://doi.org/10.1109/ICST.2014.39
28. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proc. ICSE, pp. 416–426. IEEE (2007). https://doi.org/10.1109/ICSE.2007.41
29. Margaria, T., Nagel, R., Steffen, B.: jETI: A tool for remote tool integration. In: Proc. TACAS, LNCS, vol. 3440, pp. 557–562. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_38
30. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: Proc. ISSTA, pp. 322–332. ACM (2018). https://doi.org/10.1145/3213846.3213868
31. Qiu, R., Khurshid, S., Pasareanu, C.S., Wen, J., Yang, G.: Using test ranges to improve symbolic execution. In: Proc. NFM, LNCS, vol. 10811, pp. 416–434. Springer (2018). https://doi.org/10.1007/978-3-319-77935-5_28
32. Rushby, J.M.: An Evidential Tool Bus. In: Proc. ICFEM, LNCS, vol. 3785, p. 36. Springer (2005). https://doi.org/10.1007/11576280_3
33. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000). https://doi.org/10.1145/353323.353382
34. Song, J., Alves-Foss, J.: The DARPA cyber grand challenge: A competitor's perspective, part 2. IEEE Secur. Priv. **14**(1), 76–81 (2016). https://doi.org/10.1109/MSP.2016.14
35. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1–2), 9–30 (1997). https://doi.org/10.1007/s100090050003
36. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. NDSS. Internet Society (2016). https://doi.org/10.14722/ndss.2016.23368
37. Taneja, K., Xie, T., Tillmann, N., de Halleux, J.: eXpress: Guided path exploration for efficient regression test generation. In: Proc. ISSTA, pp. 1–11. ACM (2011). https://doi.org/10.1145/2001420.2001422
38. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**, 121–189 (1995)
39. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Proc. ISSTA, pp. 97–107. ACM (2004). https://doi.org/10.1145/1007512.1007526
40. Zhu, Z., Jiao, L., Xu, X.: Combining search-based testing and dynamic symbolic execution by evolvability metric. In: Proc. ICSME, pp. 59–68. IEEE (2018). https://doi.org/10.1109/ICSME.2018.00015

Difference Verification with Conditions

Dirk Beyer[1], Marie-Christine Jakobs[1,2],
and Thomas Lemberger[1]

[1] LMU Munich, Munich, Germany
[2] Department of Computer Science, TU Darmstadt, Darmstadt, Germany

Abstract. Modern software-verification tools need to support development processes that involve frequent changes. Existing approaches for incremental verification hard-code specific verification techniques. Some of the approaches must be tightly intertwined with the development process. To solve this open problem, we present the concept of *difference verification with conditions*. Difference verification with conditions is independent from any specific verification technique and can be integrated in software projects at any time. It first applies a change analysis that detects which parts of a software were changed between revisions and encodes that information in a condition. Based on this condition, an off-the-shelf verifier is used to verify only those parts of the software that are influenced by the changes. As a proof of concept, we propose a simple, syntax-based change analysis and use difference verification with conditions with three off-the-shelf verifiers. An extensive evaluation shows the competitiveness of difference verification with conditions.

1 Introduction

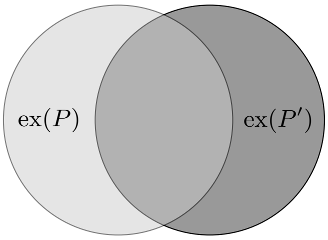
Software changes frequently during its life-cycle: developers fix bugs, adapt existing features, or add new features. In agile development, software construction is an intrinsically incremental process. Every change to a working system holds a risk to introduce a new defect. Since software failures are often costly and may even endanger human lives, it is an integral part of software development to find potential failures and ensure their absence.

However, running a full verification after each change is inadequate: Changes rarely affect the complete program behavior. For example, consider program `absSum` (Fig. 1, middle). If the assignment of program variable `r` is changed in the else-branch at location 5 ($\texttt{absSum}^{\texttt{mod}}$, Fig. 1, right), only program executions that take that else-branch show different behavior. Program executions that take the if-branch (highlighted in gray) are not affected by the change. This is typical for program changes: A modified program P' exhibits some new or changed program executions compared to an original program P, but some executions also stay the same (Fig. 1, left). To ensure the safety of P', it is sufficient to inspect only the changed behavior $\mathrm{ex}(P') \setminus \mathrm{ex}(P)$.

Replication package available on Zenodo [12].
Funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 418257054 (Coop) and 378803395 (GRK ConVeY).

© The Author(s) 2020

F. de Boer and A. Cerone (Eds.): SEFM 2020, LNCS 12310, pp. 133–154, 2020.
https://doi.org/10.1007/978-3-030-58768-0_8



```
0 r=0;
1 if(a<0)
2   while(a<0)
3     r=r-a;
4     a=a+1;
  else
5   r=a+a+1;
6   r=r/2;
```

```
0 r=0;
1 if(a<0)
2   while(a<0)
3     r=r-a;
4     a=a+1;
  else
5   r=a*(a+1);
6   r=r/2;
```

Fig. 1. Relation between program executions of original and modified program (left) and an example: Program `absSum` (middle) and its modified version `absSum`$^{\mathtt{mod}}$ (right). The modification at location 5 is shown in blue. Program parts unaffected by the modification are highlighted in gray.

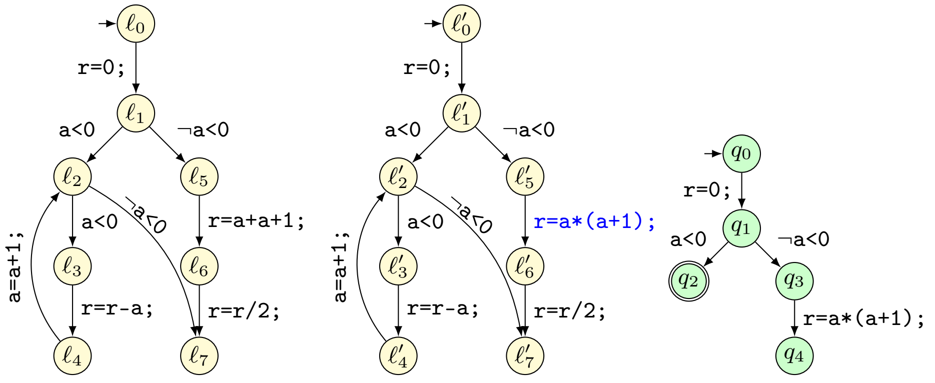
Many incremental verification approaches [39,40] use this insight: Regression-test selection [62] tries to only execute those tests in a test suite that are relevant w.r.t. the change, and incremental formal verification techniques adapt existing proofs [33,49,53,54], reuse intermediate results [16,59], or skip the exploration of unchanged behavior [21,47,60,61]. However, they (a) all focus on one fixed verification approach, (b) require a strong coupling between the original verification approach and the incremental technique, and (c) require an initial, full verification run. Often, this inflexibility makes an approach prohibitive.

As an alternative, we define the concept of *difference verification with conditions*: Given the original and the changed software, difference verification with conditions first identifies all executions that are affected by changes and encodes them in a condition, an exchange format already known from conditional model checking [10]—we call this first part DIFFCOND. Then, a conditional verifier uses that condition to verify only the changed program behavior. For this step, any existing off-the-shelf verifier can be turned into a conditional verifier with the reducer-based approach [13].

Difference verification with conditions allows us to (a) use varying verification approaches for incremental verification, (b) automatically turn any existing verifier into an incremental verifier, and (c) skip an initial, costly verification run.

Contributions. We make the following contributions:

- We propose *difference verification with conditions*, which is an incremental verification approach that combines existing tools and approaches.
- We provide the algorithm DIFFCOND, an integral part of difference verification with conditions, which outputs a description of the modified execution paths in an exchangeable condition format. We also prove its correctness.
- We implemented DIFFCOND in the verification framework CPACHECKER and combined it with existing verifiers to construct difference verifiers.
- To study the effectiveness and efficiency of difference verification with conditions, we performed an extensive evaluation on more than 10 000 C programs.
- DIFFCOND and all our data are available for replication and to construct further difference verifiers (see Sect. 7).



(a) Original program (b) Modified program (c) Condition

Fig. 2. CFA of `absSum` (Fig. 1), CFA of `absSum`$^{\texttt{mod}}$, and a condition that describes the common executions of both programs, as created by our approach

2 Background

Programs. For ease of presentation, we consider imperative programs with deterministic control-flow, which execute statements from a set *Ops*. Our implementation supports C programs. Following literature [8,9,30], we model programs as control-flow automata.

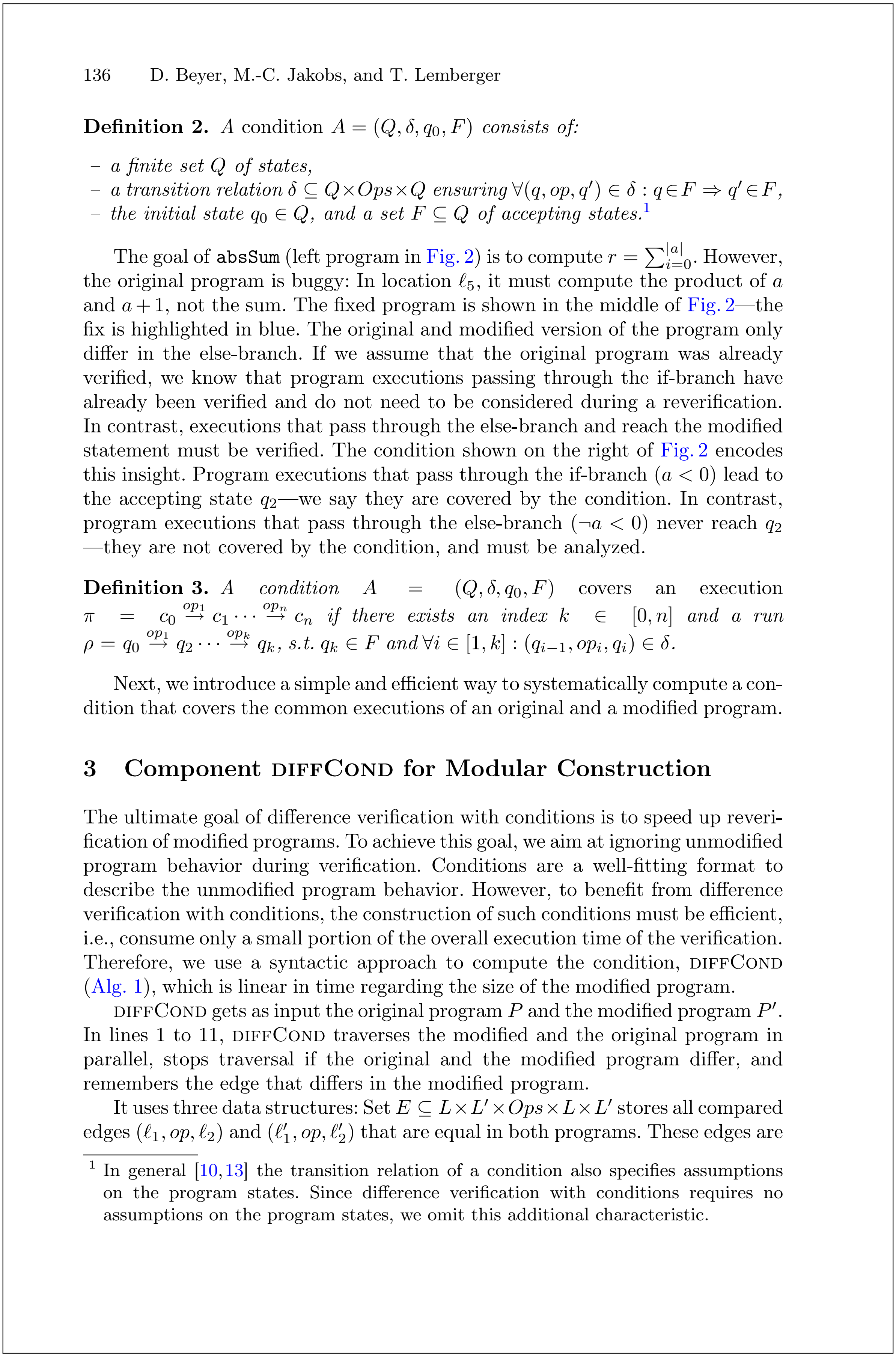
Definition 1. *A* control-flow automaton *(CFA)* $P = (L, \ell_0, G)$ *consists of*

- *a set* L *of program locations with initial location* $\ell_0 \in L$*, and*
- *a set* $G \subseteq L \times Ops \times L$ *of control-flow edges.*

CFA P *is* deterministic *if* $(\ell, op, \ell'), (\ell, op, \ell'') \in G \;\Rightarrow\; \ell' = \ell''$.

Figure 2 shows the CFA of the example program `absSum` from Fig. 1. A sequence $\ell_0 \overset{op_1}{\to} \ell_1 \cdots \overset{op_n}{\to} \ell_n$ is a *syntactical path* through CFA $P = (L, \ell_0, G)$, if $\forall i \in [1, n] : (\ell_{i-1}, op_i, \ell_i) \in G$. We rely on standard operational semantics and model a program state by a pair of (1) the program counter, whose value refers to a program location in the CFA, and (2) a concrete data state c, whose shape we do not further specify [8]. We denote the set of all concrete data states as C. The function $sp_{op} : C \to 2^C$ describes the possible effects of operation $op \in Ops$ on concrete data state $c \in C$. Based on this, a sequence $(\ell_0, c_0) \overset{op_1}{\to} (\ell_1, c_1) \cdots \overset{op_n}{\to} (\ell_n, c_n)$ is a *program path* through CFA $P = (L, \ell_0, G)$, if $\ell_0 \overset{op_1}{\to} \ell_1 \cdots \overset{op_n}{\to} \ell_n$ is a syntactical path through P and $\forall i \in [1, n] : c_i \in sp_{op_i}(c_{i-1})$. We denote the set of all program paths by $\mathrm{paths}(P)$. *Program executions* are derived from program paths. If $p = (\ell_0, c_0) \overset{op_1}{\to} (\ell_1, c_1) \cdots \overset{op_n}{\to} (\ell_n, c_n)$ is a program path, then $\mathrm{ex}(p) = c_0 \overset{op_1}{\to} c_1 \cdots \overset{op_n}{\to} c_n$ is a program execution. The executions of a program P are defined as $\mathrm{ex}(P) := \{ex(p) \mid p \in \mathrm{paths}(P)\}$.

Conditions. A condition describes which program executions were already verified, e.g., in a previous verification run. We use automata to represent conditions and use accepting states to identify already verified executions [13].



Definition 2. *A* condition $A = (Q, \delta, q_0, F)$ *consists of:*

- *a finite set* Q *of states,*
- *a transition relation* $\delta \subseteq Q \times Ops \times Q$ *ensuring* $\forall (q, op, q') \in \delta : q \in F \Rightarrow q' \in F$,
- *the initial state* $q_0 \in Q$, *and a set* $F \subseteq Q$ *of accepting states.*[1]

The goal of `absSum` (left program in Fig. 2) is to compute $r = \sum_{i=0}^{|a|}$. However, the original program is buggy: In location ℓ_5, it must compute the product of a and $a+1$, not the sum. The fixed program is shown in the middle of Fig. 2—the fix is highlighted in blue. The original and modified version of the program only differ in the else-branch. If we assume that the original program was already verified, we know that program executions passing through the if-branch have already been verified and do not need to be considered during a reverification. In contrast, executions that pass through the else-branch and reach the modified statement must be verified. The condition shown on the right of Fig. 2 encodes this insight. Program executions that pass through the if-branch ($a < 0$) lead to the accepting state q_2—we say they are covered by the condition. In contrast, program executions that pass through the else-branch ($\neg a < 0$) never reach q_2—they are not covered by the condition, and must be analyzed.

Definition 3. *A condition* $A = (Q, \delta, q_0, F)$ covers *an* execution $\pi = c_0 \xrightarrow{op_1} c_1 \cdots \xrightarrow{op_n} c_n$ *if there exists an index* $k \in [0, n]$ *and a run* $\rho = q_0 \xrightarrow{op_1} q_2 \cdots \xrightarrow{op_k} q_k$, *s.t.* $q_k \in F$ *and* $\forall i \in [1, k] : (q_{i-1}, op_i, q_i) \in \delta$.

Next, we introduce a simple and efficient way to systematically compute a condition that covers the common executions of an original and a modified program.

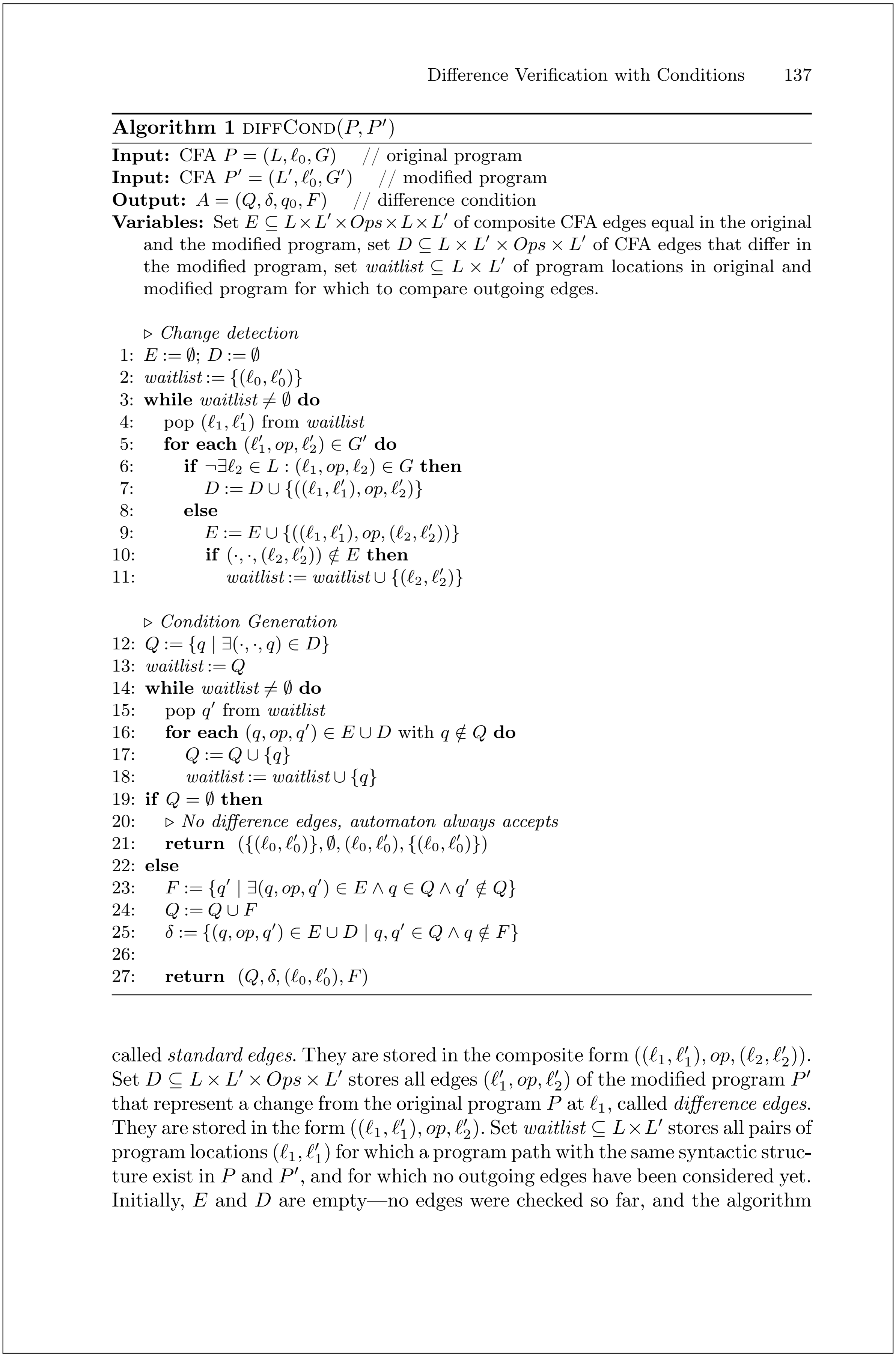
3 Component diffCond for Modular Construction

The ultimate goal of difference verification with conditions is to speed up reverification of modified programs. To achieve this goal, we aim at ignoring unmodified program behavior during verification. Conditions are a well-fitting format to describe the unmodified program behavior. However, to benefit from difference verification with conditions, the construction of such conditions must be efficient, i.e., consume only a small portion of the overall execution time of the verification. Therefore, we use a syntactic approach to compute the condition, diffCond (Alg. 1), which is linear in time regarding the size of the modified program.

diffCond gets as input the original program P and the modified program P'. In lines 1 to 11, diffCond traverses the modified and the original program in parallel, stops traversal if the original and the modified program differ, and remembers the edge that differs in the modified program.

It uses three data structures: Set $E \subseteq L \times L' \times Ops \times L \times L'$ stores all compared edges (ℓ_1, op, ℓ_2) and (ℓ'_1, op, ℓ'_2) that are equal in both programs. These edges are

[1] In general [10,13] the transition relation of a condition also specifies assumptions on the program states. Since difference verification with conditions requires no assumptions on the program states, we omit this additional characteristic.



Algorithm 1 $\textsc{diffCond}(P, P')$

Input: CFA $P = (L, \ell_0, G)$ // original program
Input: CFA $P' = (L', \ell'_0, G')$ // modified program
Output: $A = (Q, \delta, q_0, F)$ // difference condition
Variables: Set $E \subseteq L \times L' \times \mathit{Ops} \times L \times L'$ of composite CFA edges equal in the original and the modified program, set $D \subseteq L \times L' \times \mathit{Ops} \times L'$ of CFA edges that differ in the modified program, set *waitlist* $\subseteq L \times L'$ of program locations in original and modified program for which to compare outgoing edges.

```
▷ Change detection
$E := \emptyset$; $D := \emptyset$
$\mathit{waitlist} := \{(\ell_0, \ell'_0)\}$
while $\mathit{waitlist} \neq \emptyset$ do
    pop $(\ell_1, \ell'_1)$ from $\mathit{waitlist}$
    for each $(\ell'_1, op, \ell'_2) \in G'$ do
        if $\neg\exists \ell_2 \in L : (\ell_1, op, \ell_2) \in G$ then
            $D := D \cup \{((\ell_1, \ell'_1), op, \ell'_2)\}$
        else
            $E := E \cup \{((\ell_1, \ell'_1), op, (\ell_2, \ell'_2))\}$
            if $(\cdot, \cdot, (\ell_2, \ell'_2)) \notin E$ then
                $\mathit{waitlist} := \mathit{waitlist} \cup \{(\ell_2, \ell'_2)\}$

▷ Condition Generation
$Q := \{q \mid \exists(\cdot, \cdot, q) \in D\}$
$\mathit{waitlist} := Q$
while $\mathit{waitlist} \neq \emptyset$ do
    pop $q'$ from $\mathit{waitlist}$
    for each $(q, op, q') \in E \cup D$ with $q \notin Q$ do
        $Q := Q \cup \{q\}$
        $\mathit{waitlist} := \mathit{waitlist} \cup \{q\}$
if $Q = \emptyset$ then
    ▷ No difference edges, automaton always accepts
    return $(\{(\ell_0, \ell'_0)\}, \emptyset, (\ell_0, \ell'_0), \{(\ell_0, \ell'_0)\})$
else
    $F := \{q' \mid \exists(q, op, q') \in E \wedge q \in Q \wedge q' \notin Q\}$
    $Q := Q \cup F$
    $\delta := \{(q, op, q') \in E \cup D \mid q, q' \in Q \wedge q \notin F\}$

    return $(Q, \delta, (\ell_0, \ell'_0), F)$
```

called *standard edges*. They are stored in the composite form $((\ell_1, \ell'_1), op, (\ell_2, \ell'_2))$. Set $D \subseteq L \times L' \times \mathit{Ops} \times L'$ stores all edges (ℓ'_1, op, ℓ'_2) of the modified program P' that represent a change from the original program P at ℓ_1, called *difference edges*. They are stored in the form $((\ell_1, \ell'_1), op, \ell'_2)$. Set *waitlist* $\subseteq L \times L'$ stores all pairs of program locations (ℓ_1, ℓ'_1) for which a program path with the same syntactic structure exist in P and P', and for which no outgoing edges have been considered yet. Initially, E and D are empty—no edges were checked so far, and the algorithm

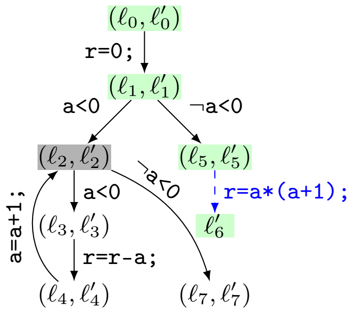


Fig. 3. Parallel composition of `absSum` and `absSum`$^{\texttt{mod}}$ as computed by DIFFCOND

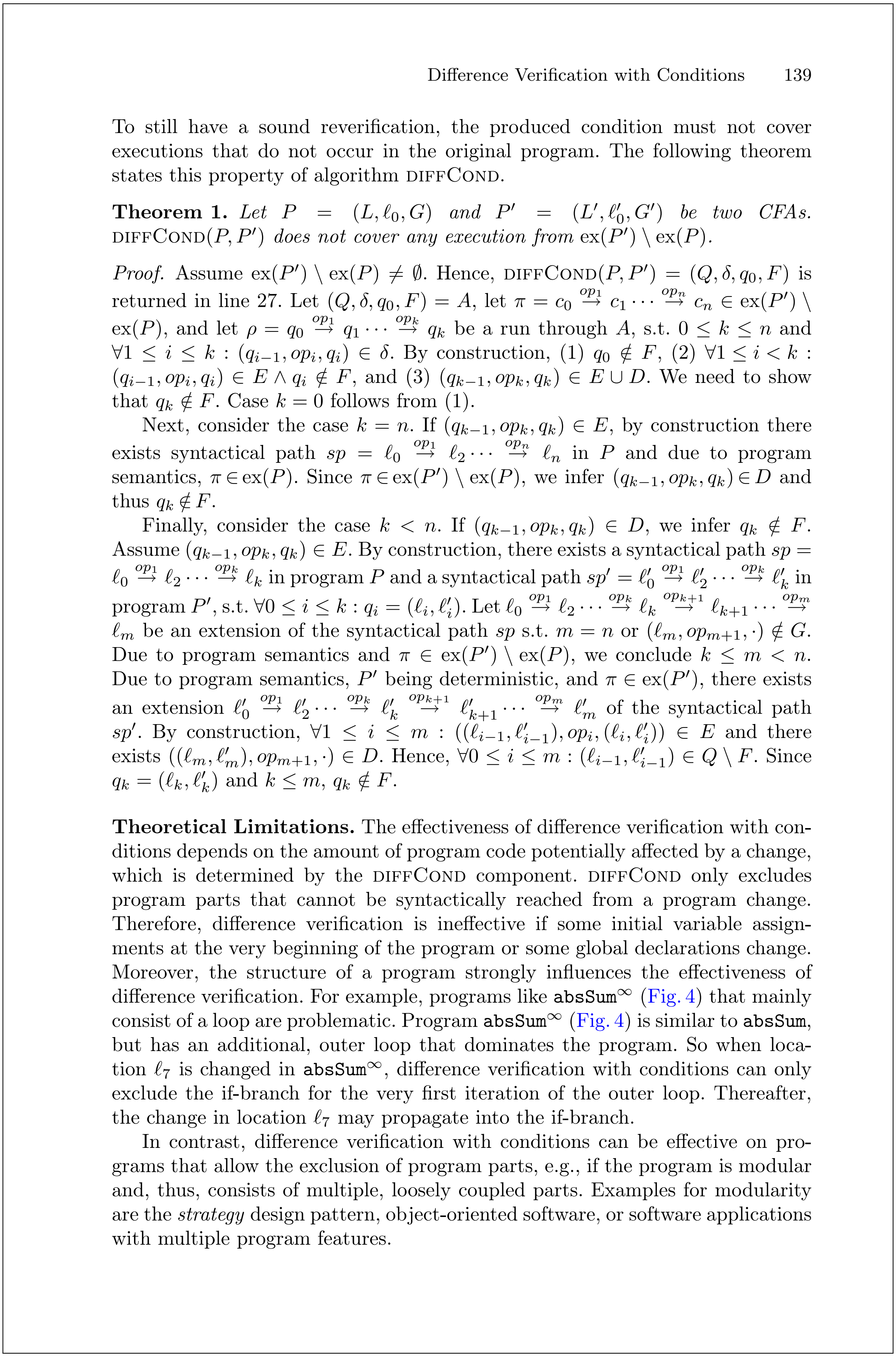
starts at the two initial program locations, i.e., *waitlist* $= \{(\ell_0, \ell'_0)\}$ (lines 1 and 2). As long as *waitlist* contains program locations, the algorithm picks one of them, here depicted as (ℓ_1, ℓ'_1) (line 4). It considers all outgoing edges (ℓ'_1, op, ℓ'_2) of ℓ'_1 in the modified program. If the same operation *op* does not exist at any outgoing edge of ℓ_1, it is considered to be changed and the difference edge $((\ell_1, \ell'_1), op, \ell'_2)$ is stored in D before continuing with the next state in *waitlist*. However, if the same operation *op* exists at an outgoing edge (ℓ_1, op, ℓ_2), it is considered to be equal and the standard edge $((\ell_1, \ell'_1), op, (\ell_2, \ell'_2))$ is stored in E before continuing with the next state in *waitlist*. To this end, DIFFCOND explores the syntactical composition of the original and modified program. In addition, if the tuple (ℓ_2, ℓ'_2) of locations has not been detected before (line 10), it is added to the waitlist for further exploration. Figure 3 shows the graph built from edges E (black) and D (blue and dashed) when executing DIFFCOND on `absSum` and `absSum`$^{\texttt{mod}}$.

To compute the condition, we first determine the condition's states. Lines 12 to 18 compute all nodes that can reach a successor of a difference edge. Figure 3 highlights these nodes in green. Nodes that are not discovered in lines 12–18 cannot lead to a difference edge and, thus, not to different program behavior. Consequently, undiscovered nodes that are successors of nodes discovered in lines 12–18 become final states (line 23). Figure 3 highlights these nodes in gray (only node (ℓ_2, ℓ'_2)). The union of discovered and final states become our condition states. To complete the construction, we use the pair of initial program locations as the initial state (ℓ_0, ℓ'_0) and add to the transition relation all transitions from E and D that connect condition states. Figure 2c shows the condition created from Fig. 3.

Finally, note that lines 19–21 handle the special case that the set D of difference edges is empty, thus resulting in $Q = \emptyset$ in line 19. The set D is empty if the original and the modified program only differ in the names of their program locations[2] or if the modified program is empty $((\ell'_0, \cdot, \cdot) \notin G')$. In both cases, all executions of the modified program are covered by the executions of the original program. As a result, the condition covers all executions: its only state is both initial and accepting state, and the condition has no transitions.

The purpose of algorithm DIFFCOND is to compute a condition that supports skipping unchanged behavior during reverification of a modified program.

[2] In practice, this can happen if empty lines are added or removed from the program.



To still have a sound reverification, the produced condition must not cover executions that do not occur in the original program. The following theorem states this property of algorithm diffCond.

Theorem 1. *Let* $P = (L, \ell_0, G)$ *and* $P' = (L', \ell'_0, G')$ *be two CFAs.* diffCond(P, P') *does not cover any execution from* $\mathrm{ex}(P') \setminus \mathrm{ex}(P)$*.*

Proof. Assume $\mathrm{ex}(P') \setminus \mathrm{ex}(P) \neq \emptyset$. Hence, diffCond$(P, P') = (Q, \delta, q_0, F)$ is returned in line 27. Let $(Q, \delta, q_0, F) = A$, let $\pi = c_0 \xrightarrow{op_1} c_1 \cdots \xrightarrow{op_n} c_n \in \mathrm{ex}(P') \setminus \mathrm{ex}(P)$, and let $\rho = q_0 \xrightarrow{op_1} q_1 \cdots \xrightarrow{op_k} q_k$ be a run through A, s.t. $0 \leq k \leq n$ and $\forall 1 \leq i \leq k : (q_{i-1}, op_i, q_i) \in \delta$. By construction, (1) $q_0 \notin F$, (2) $\forall 1 \leq i < k : (q_{i-1}, op_i, q_i) \in E \wedge q_i \notin F$, and (3) $(q_{k-1}, op_k, q_k) \in E \cup D$. We need to show that $q_k \notin F$. Case $k = 0$ follows from (1).

Next, consider the case $k = n$. If $(q_{k-1}, op_k, q_k) \in E$, by construction there exists syntactical path $sp = \ell_0 \xrightarrow{op_1} \ell_2 \cdots \xrightarrow{op_n} \ell_n$ in P and due to program semantics, $\pi \in \mathrm{ex}(P)$. Since $\pi \in \mathrm{ex}(P') \setminus \mathrm{ex}(P)$, we infer $(q_{k-1}, op_k, q_k) \in D$ and thus $q_k \notin F$.

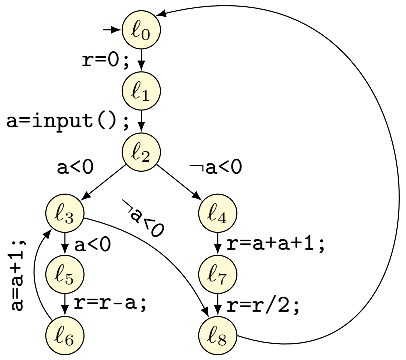
Finally, consider the case $k < n$. If $(q_{k-1}, op_k, q_k) \in D$, we infer $q_k \notin F$. Assume $(q_{k-1}, op_k, q_k) \in E$. By construction, there exists a syntactical path $sp = \ell_0 \xrightarrow{op_1} \ell_2 \cdots \xrightarrow{op_k} \ell_k$ in program P and a syntactical path $sp' = \ell'_0 \xrightarrow{op_1} \ell'_2 \cdots \xrightarrow{op_k} \ell'_k$ in program P', s.t. $\forall 0 \leq i \leq k : q_i = (\ell_i, \ell'_i)$. Let $\ell_0 \xrightarrow{op_1} \ell_2 \cdots \xrightarrow{op_k} \ell_k \xrightarrow{op_{k+1}} \ell_{k+1} \cdots \xrightarrow{op_m} \ell_m$ be an extension of the syntactical path sp s.t. $m = n$ or $(\ell_m, op_{m+1}, \cdot) \notin G$. Due to program semantics and $\pi \in \mathrm{ex}(P') \setminus \mathrm{ex}(P)$, we conclude $k \leq m < n$. Due to program semantics, P' being deterministic, and $\pi \in \mathrm{ex}(P')$, there exists an extension $\ell'_0 \xrightarrow{op_1} \ell'_2 \cdots \xrightarrow{op_k} \ell'_k \xrightarrow{op_{k+1}} \ell'_{k+1} \cdots \xrightarrow{op_m} \ell'_m$ of the syntactical path sp'. By construction, $\forall 1 \leq i \leq m : ((\ell_{i-1}, \ell'_{i-1}), op_i, (\ell_i, \ell'_i)) \in E$ and there exists $((\ell_m, \ell'_m), op_{m+1}, \cdot) \in D$. Hence, $\forall 0 \leq i \leq m : (\ell_{i-1}, \ell'_{i-1}) \in Q \setminus F$. Since $q_k = (\ell_k, \ell'_k)$ and $k \leq m$, $q_k \notin F$.

Theoretical Limitations. The effectiveness of difference verification with conditions depends on the amount of program code potentially affected by a change, which is determined by the diffCond component. diffCond only excludes program parts that cannot be syntactically reached from a program change. Therefore, difference verification is ineffective if some initial variable assignments at the very beginning of the program or some global declarations change. Moreover, the structure of a program strongly influences the effectiveness of difference verification. For example, programs like `absSum`$^\infty$ (Fig. 4) that mainly consist of a loop are problematic. Program `absSum`$^\infty$ (Fig. 4) is similar to `absSum`, but has an additional, outer loop that dominates the program. So when location ℓ_7 is changed in `absSum`$^\infty$, difference verification with conditions can only exclude the if-branch for the very first iteration of the outer loop. Thereafter, the change in location ℓ_7 may propagate into the if-branch.

In contrast, difference verification with conditions can be effective on programs that allow the exclusion of program parts, e.g., if the program is modular and, thus, consists of multiple, loosely coupled parts. Examples for modularity are the *strategy* design pattern, object-oriented software, or software applications with multiple program features.

```
0 while (1)
1   r=0;
2   a=input();
3   if(a<0)
4     while(a<0)
5       r=r-a;
6       a=a+1;
    else
7     r=a+a+1;
8     r=r/2;
```

(a) Program code



(b) CFA of program

Fig. 4. Example program absSum$^{\infty}$ with loop dominating the whole program

When designing our experiments, we will consider these limitations of difference verification with conditions. Before we get to our experiments, we must describe the modular composition of the diffCond component with a verifier, which specifies the difference verifier.

4 Modular Combinations with Existing Verifiers

The diffCond algorithm can be combined with any off-the-shelf conditional verifier [10] to produce a difference verifier in a modular way. The goal of a difference verifier is to verify only modified program paths. To this end, it first uses diffCond to discover potentially modified program paths and then runs a conditional verifier to explore only those paths identified by diffCond. Figure 5 shows the construction template for difference verification with conditions. diffCond gets the original and modified program as input and encodes the modified paths in a condition. The constructed condition is forwarded to a conditional verifier, which uses the condition to restrict its analysis of the modified program to those paths that are not covered by the condition (i.e., the modified paths). Based on this template, we can construct difference verifiers from arbitrary conditional verifiers. Moreover, we can construct difference verifiers from non-conditional verifiers by using the concept of reducer-based conditional verifiers [13]. The idea of a reducer-based conditional verifier is shown on the right of Fig. 5. To turn an arbitrary verifier into a conditional one, a reducer-based conditional verifier puts a preprocessor (called reducer) in front of the verifier. The reducer gets a program and a condition and outputs a new, residual program that represents the program paths not covered by the condition. A full verification

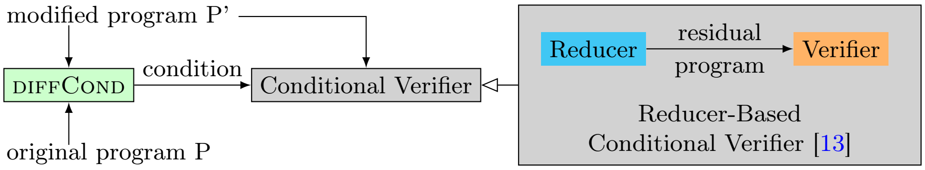


Fig. 5. diffCond + conditional verifier = difference verifier

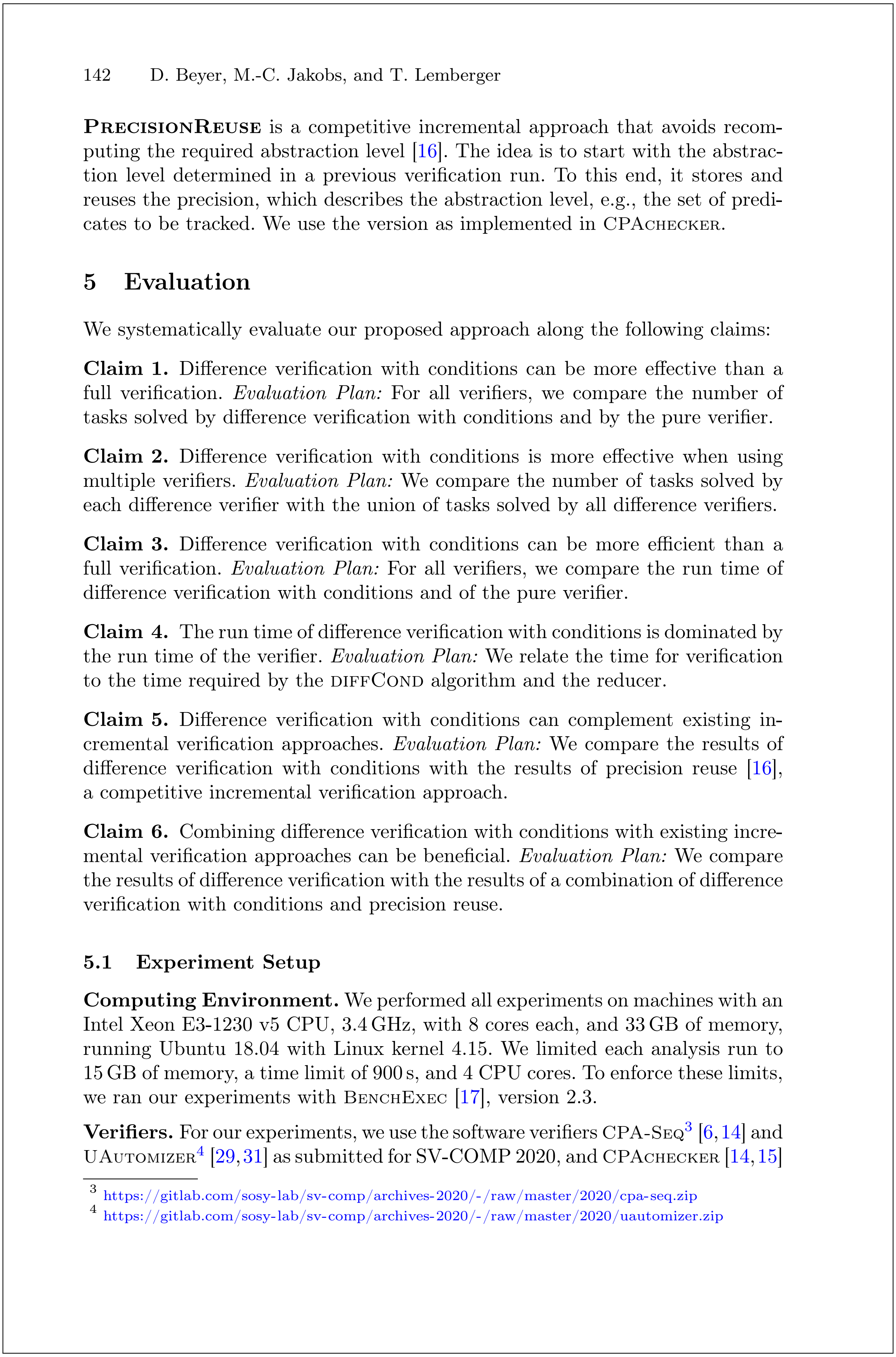
of this residual program is then equivalent to a conditional verification of the original program with the produced condition. However, note that the existing reducers are designed for model checkers and do not necessarily work with other verification technologies like deductive verifiers.

In this paper, we transform three verifiers into difference verifiers: CPA-Seq, UAutomizer, and Predicate. The first two are the best verifiers from SV-COMP 2020 [5], and the third is a predicate-abstraction approach. We use the off-the-shelf verifiers CPA-Seq and UAutomizer as non-conditional verifiers and thus add a reducer, while we use Predicate as conditional verifier. Since a difference verifier can now be built from any off-the-shelf verifier, we can also combine difference verification with other incremental verification techniques. As an example, we can use precision reuse [16]. This technique is implemented in CPAchecker [16] and UAutomizer [49] and can be used with the previously mentioned approaches. Next we explain the technologies of the selected verifiers.

CPA-Seq uses several different strategies from the CPAchecker verification framework [6,11,14]. CPA-Seq first analyzes different features of the program under verification. The program features considered are: recursion, concurrency, occurrence of loops, and occurrence of complex data types like pointers and structs. Based on these features, CPA-Seq uses one of five different verification techniques (cf. [6]). For non-recursive, non-concurrent programs with a non-trivial control flow, CPA-Seq uses a sequential combination of four different analyses: It uses value analysis with and without Counterexample-guided Abstraction Refinement (CEGAR) [24], a predicate analysis similar to Predicate, and k-induction with invariant generation [7]. Invariants are generated by numerical and predicate analyses and are forwarded to the k-induction analysis.

UAutomizer is the automata-based approach from the Ultimate verification framework [29,31]. It uses a CEGAR approach to successively refine an over-approximation of the error paths, which is given in form of automata. In each refinement step, a generalization of an infeasible error path is excluded from the over-approximation. The generalization of the error path is described by a Floyd-Hoare automaton [31], which assigns Boolean formulas over predicates to its states. The predicates are obtained via interpolation along the infeasible error path [43].

Predicate is the predicate-abstraction approach from the CPAchecker framework [14] with adjustable-block encoding (ABE) [15]. ABE is instructed to abstract at loop heads only. CEGAR together with lazy refinement [34] and interpolation [32] determines the necessary set of predicates.



PrecisionReuse is a competitive incremental approach that avoids recomputing the required abstraction level [16]. The idea is to start with the abstraction level determined in a previous verification run. To this end, it stores and reuses the precision, which describes the abstraction level, e.g., the set of predicates to be tracked. We use the version as implemented in CPAchecker.

5 Evaluation

We systematically evaluate our proposed approach along the following claims:

Claim 1. Difference verification with conditions can be more effective than a full verification. *Evaluation Plan:* For all verifiers, we compare the number of tasks solved by difference verification with conditions and by the pure verifier.

Claim 2. Difference verification with conditions is more effective when using multiple verifiers. *Evaluation Plan:* We compare the number of tasks solved by each difference verifier with the union of tasks solved by all difference verifiers.

Claim 3. Difference verification with conditions can be more efficient than a full verification. *Evaluation Plan:* For all verifiers, we compare the run time of difference verification with conditions and of the pure verifier.

Claim 4. The run time of difference verification with conditions is dominated by the run time of the verifier. *Evaluation Plan:* We relate the time for verification to the time required by the diffCond algorithm and the reducer.

Claim 5. Difference verification with conditions can complement existing incremental verification approaches. *Evaluation Plan:* We compare the results of difference verification with conditions with the results of precision reuse [16], a competitive incremental verification approach.

Claim 6. Combining difference verification with conditions with existing incremental verification approaches can be beneficial. *Evaluation Plan:* We compare the results of difference verification with the results of a combination of difference verification with conditions and precision reuse.

5.1 Experiment Setup

Computing Environment. We performed all experiments on machines with an Intel Xeon E3-1230 v5 CPU, 3.4 GHz, with 8 cores each, and 33 GB of memory, running Ubuntu 18.04 with Linux kernel 4.15. We limited each analysis run to 15 GB of memory, a time limit of 900 s, and 4 CPU cores. To enforce these limits, we ran our experiments with BenchExec [17], version 2.3.

Verifiers. For our experiments, we use the software verifiers CPA-Seq[3] [6,14] and UAutomizer[4] [29,31] as submitted for SV-COMP 2020, and CPAchecker [14,15]

[3] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/raw/master/2020/cpa-seq.zip

[4] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/raw/master/2020/uautomizer.zip

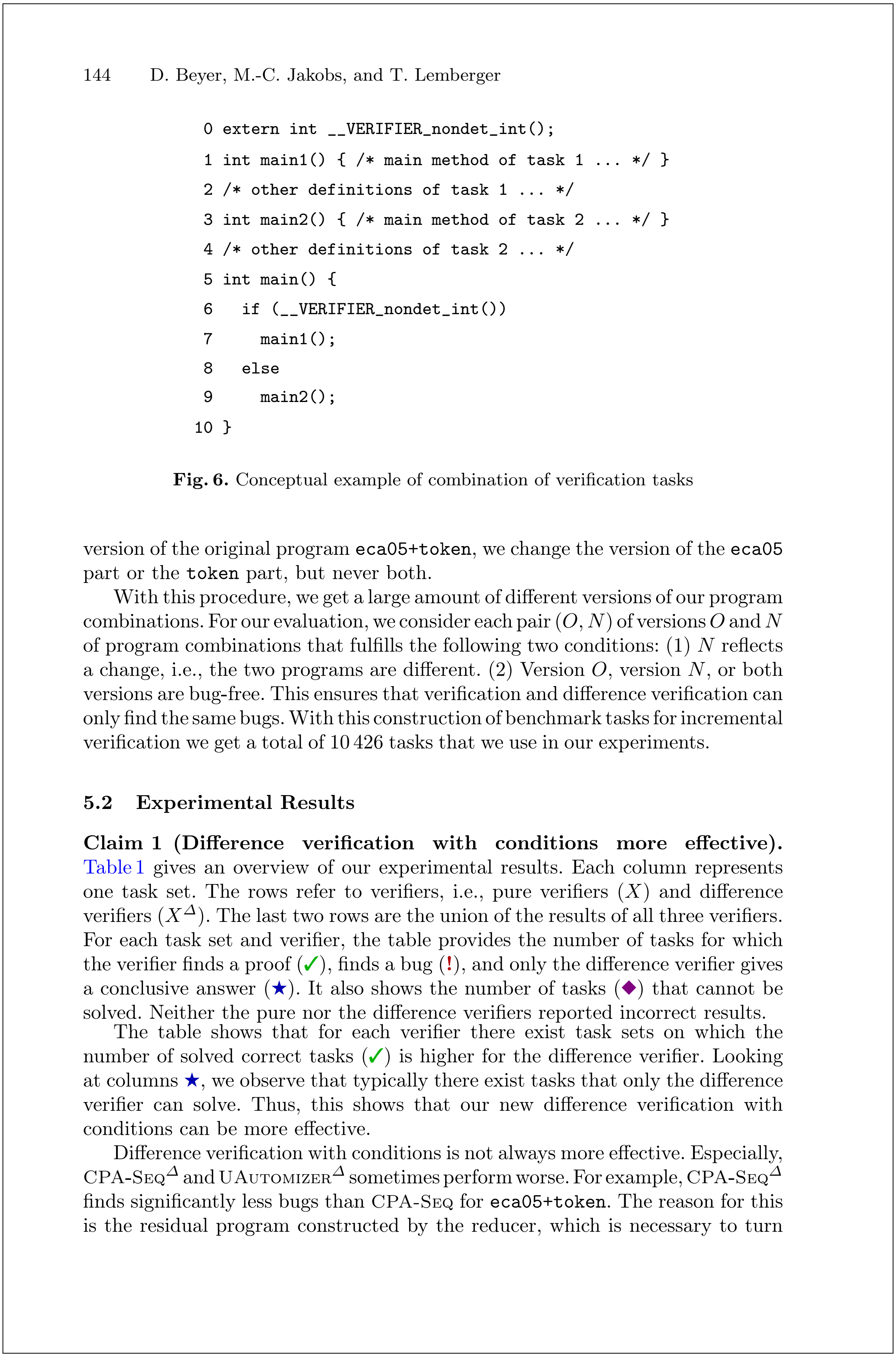
in revision 32864[5]. CPA-Seq and UAutomizer are used as verifiers. CPAchecker provides the verifier Predicate, but also the new diffCond component and the Reducer component for reducer-based conditional verification. The difference verifier based on Predicate is realized as a single run. In contrast, the difference verifiers based on CPA-Seq and UAutomizer are realized as composition of two separate runs. The first run executes the diffCond algorithm followed by the reducer to generate the residual program. It is only executed once per task, i.e., the same residual programs are given to CPA-Seq and UAutomizer. In a second run, CPA-Seq and UAutomizer, respectively, verify the residual program. To deal with residual programs, we increased the Java stack size for CPA-Seq and UAutomizer.

Existing Incremental Verifier. We use Predicate with precision reuse [16].

Verification Tasks. We use verification tasks from the public repository `sv-benchmarks` (tag `svcomp20`)[6], which is the most diverse, largest, and well-established collection of verification tasks. Since difference verification with conditions is an incremental verification approach, we require different program versions. We searched the benchmark repository for programs that come with multiple versions and for which at least one version is hard to solve, i.e., at least one of the three considered verifiers takes more than 100 s for verification of that version, but is successful. From these programs, we arbitrarily picked the following: `eca05` and `eca12` (event-condition-action systems, both have 10 versions each), `gcd` (greatest common divisor computation, has 4 versions), `newton` (approximation of sine, has 24 versions), `pals` (leader election, has 26 versions), `sfifo` (second-chance FIFO replacement, has 5 versions), `softflt` (a software implementation of floats, has 5 versions), `square` (square-root computation, has 8 versions), and `token` (a communication protocol, has 28 versions). Unfortunately, all of these programs are specialized implementations with a single purpose. Thus, their implementation is strongly coupled and any reasonable program change affects the complete program. As explained before, this prohibits effective difference verification with conditions.

To get benchmark tasks that instead contain independent program parts, we create new combinations from the selected programs. We choose two programs, e.g., `eca05` and `token`. We then combine these two programs according to the following scheme: We create a new program with all declarations and definitions of both original programs, but a new main function. This new main function randomly calls the main function of one of the two original programs. Name clashes are resolved via renaming. Figure 6 shows the conceptual structure of each program created through this combination. For our experiments, we consider the following combinations of programs: (1) eca05+token, (2) gcd+newton, (3) pals+eca12, (4) sfifo+token, (5) square+softflt. To create different versions of our combinations, we replace one of the two program parts with a different version of that part. For example, to get a different

[5] https://gitlab.com/sosy-lab/software/cpachecker/-/tree/230d2ca5
[6] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20



```
extern int __VERIFIER_nondet_int();
int main1() { /* main method of task 1 ... */ }
/* other definitions of task 1 ... */
int main2() { /* main method of task 2 ... */ }
/* other definitions of task 2 ... */
int main() {
  if (__VERIFIER_nondet_int())
    main1();
  else
    main2();
}
```

Fig. 6. Conceptual example of combination of verification tasks

version of the original program `eca05+token`, we change the version of the `eca05` part or the `token` part, but never both.

With this procedure, we get a large amount of different versions of our program combinations. For our evaluation, we consider each pair (O, N) of versions O and N of program combinations that fulfills the following two conditions: (1) N reflects a change, i.e., the two programs are different. (2) Version O, version N, or both versions are bug-free. This ensures that verification and difference verification can only find the same bugs. With this construction of benchmark tasks for incremental verification we get a total of 10 426 tasks that we use in our experiments.

5.2 Experimental Results

Claim 1 (Difference verification with conditions more effective). Table 1 gives an overview of our experimental results. Each column represents one task set. The rows refer to verifiers, i.e., pure verifiers (X) and difference verifiers (X^{Δ}). The last two rows are the union of the results of all three verifiers. For each task set and verifier, the table provides the number of tasks for which the verifier finds a proof (✓), finds a bug (!), and only the difference verifier gives a conclusive answer (★). It also shows the number of tasks (◆) that cannot be solved. Neither the pure nor the difference verifiers reported incorrect results.

The table shows that for each verifier there exist task sets on which the number of solved correct tasks (✓) is higher for the difference verifier. Looking at columns ★, we observe that typically there exist tasks that only the difference verifier can solve. Thus, this shows that our new difference verification with conditions can be more effective.

Difference verification with conditions is not always more effective. Especially, CPA-Seq$^{\Delta}$ and UAutomizer$^{\Delta}$ sometimes perform worse. For example, CPA-Seq$^{\Delta}$ finds significantly less bugs than CPA-Seq for `eca05+token`. The reason for this is the residual program constructed by the reducer, which is necessary to turn

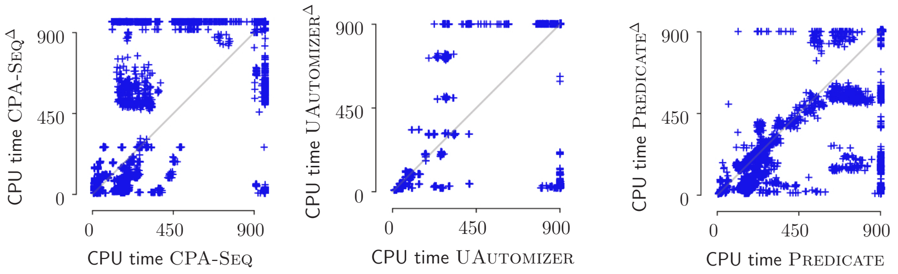
Table 1. Experimental results for Predicate, CPA-Seq and UAutomizer, as pure verifiers (X) and difference verifiers (X^{Δ}) showing how many correct tasks (✓) and tasks with a bug (!) are solved, how many tasks are only solved by the difference verifier (★) and which are too hard to solve (◆)

	eca05+token (3 640)				gcd+newton (1 924)				pals+eca12 (2 750)				sfifo+token (1 872)				square+softflt (240)			
	✓	!	★	◆	✓	!	★	◆	✓	!	★	◆	✓	!	★	◆	✓	!	★	◆
Predicate$^{\Delta}$	1447	999	451	1194	48	572	48	1304	15	55	20	2680	655	494	98	723	81	75	70	84
Predicate	1080	944		1616	0	572		1352	0	50		2700	558	507		807	33	53		154
CPA-Seq$^{\Delta}$	966	671	350	2003	48	572	48	1304	183	50	233	2517	480	390	108	1002	61	69	61	110
CPA-Seq	755	1268		1617	0	572		1352	0	0		2750	372	619		881	0	75		165
UAutomizer$^{\Delta}$	270	260	270	3110	16	0	16	1908	0	0	0	2750	349	234	112	1289	61	45	49	134
UAutomizer	0	325		3315	0	520		1404	0	48		2702	341	258		1273	44	57		139
All$^{\Delta}$	1527	999	448	1114	48	572	48	1304	183	95	228	2472	655	494	98	723	81	75	40	84
All	1080	1295		1265	0	572		1352	0	50		2700	558	626		688	55	75		110

CPA-Seq into the required conditional verifier. The created residual programs, on which the off-the-shelf verifiers run, have a different structure than the original program. They make heavy use of goto statements and deeply nested branching structures. While semantically equivalent, this can have unexpected effects on analyses: In the case of the tasks in `eca05+token`, CPA-Seq was not able to detect required information about loops and thus aborts its verification. Note that this is not a direct issue of difference verification with conditions, but an orthogonal issue. To fix the problem, verification tools must be improved to better deal with the generated residual programs or the structure of the residual program must be improved. Despite of the problem with residual programs, difference verification can solve many tasks that a full verification run cannot solve.

Since Predicate is already a conditional model checker, Predicate$^{\Delta}$ does not suffer from the residual program problem. Thus, the effectiveness of difference verification with conditions becomes even more obvious when comparing Predicate with Predicate$^{\Delta}$. For the first three task sets, Predicate$^{\Delta}$ solves all tasks that Predicate solves plus a significant amount of additional tasks that Predicate cannot solve. For the last two task sets Predicate$^{\Delta}$ fails to solve a few tasks that Predicate can solve. However, Predicate$^{\Delta}$ still solves more tasks in total. One reason for this is that the predicate abstraction used by Predicate may compute different predicates (due to a slightly different exploration of the state space), which may result in a more expensive abstraction, if the explored state-space looks different. For some tasks, these different predicates may be less suited to solve the task and thus require more time, which results in the analysis hitting the time limit. Typically, we observe this phenomenon when Predicate is expensive already (in our experiments, when it takes at least 700 s). While for complicated tasks with large changes, difference verification may produce worse results, Predicate$^{\Delta}$ is still more effective than Predicate in all categories.

Claim 2 (Better with several verifiers). To study the usefulness of using several verifiers in difference verification, we look at the tasks solved by the three difference verifiers together. We observe that Predicate$^{\Delta}$ solves the most tasks in all task sets except for `pals+eca12`, in which CPA-Seq$^{\Delta}$ is better. Moreover,



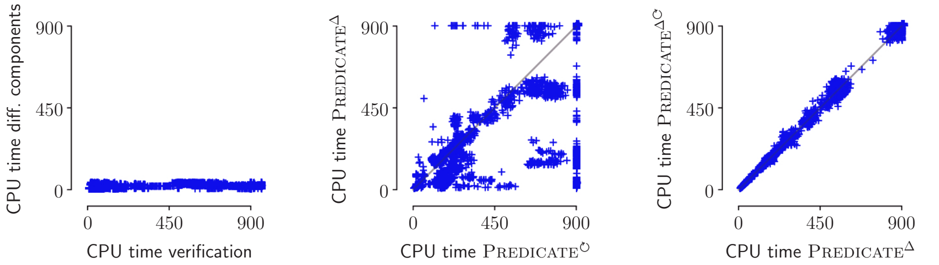
(a) CPA-SEQ, CPA-SEQ$^{\Delta}$ (b) UAUTOMIZER, UAUTOMIZER$^{\Delta}$ (c) PREDICATE, PREDICATE$^{\Delta}$

Fig. 7. CPU time (in s) of full verification vs. difference verification, per task

when looking at All$^{\Delta}$, which takes the union of all results, we observe that for `eca05+token` multiple tasks without a property violation exist that cannot be solved by the best difference verifier of this task set (PREDICATE$^{\Delta}$). Thus, the difference verification is more effective when using several verifiers.

Claim 3 (Difference verification with conditions more efficient). We compare the run times of the verifiers with the run times of the difference verifiers. For all three verifiers, the scatter plots in Fig. 7 show the CPU time required to check a task without (x-axis) and with difference verification (y-axis). If a task was not solved, because the verifier either runs out of resources or encountered an error, we assume the maximum CPU time of 900 s. Figures 7a and 7b compare the two non-conditional verifiers CPA-SEQ and UAUTOMIZER, for which we use the reducer-based conditional verifier approach. For a significant number of tasks (below diagonal), the difference verifier is faster than the respective verifier CPA-SEQ and UAUTOMIZER, and the tasks on the right edge can only be solved by the difference verifier. There are tasks for which difference verification is slower (above diagonal). Note that the problem is the residual program, not our approach. For example, many tasks located at the upper edge do not represent timeouts of the difference verification, but failures of the verifier caused by the structure of the residual program. Figure 7c compares the conditional verifier PREDICATE. For the majority of tasks, the CPU time required by PREDICATE$^{\Delta}$ is equal to or less than the time required by PREDICATE (tasks below the line). Moreover, there are only few tasks for which PREDICATE$^{\Delta}$ is slower than PREDICATE (tasks above the line). The reason for this slow-down is most likely the computation of worse predicates (see Claim 1). To sum up, difference verification with conditions can successfully increase efficiency.

Claim 4 (Verifier dominates run time). We aim to show that the DIFFCOND component and the residual program construction (in the reducer-based approach to construct conditional verifiers) require a negligible run time compared to the complete verification run time. We show in Fig. 8a for each task verified with CPA-SEQ$^{\Delta}$ and UAUTOMIZER$^{\Delta}$, the CPU time required by the full verification run (x-axis) and the CPU time of that run spent for DIFFCOND plus the reducer (y-axis). The time required by DIFFCOND + reducer does not depend on the run time of the verifier, and it is below 60 s for all tasks.



(a) Full verification, diffCond + reducer (b) Predicate$^{\circlearrowright}$, Predicate$^{\Delta}$ (c) Predicate$^{\Delta}$, Predicate$^{\Delta\circlearrowright}$

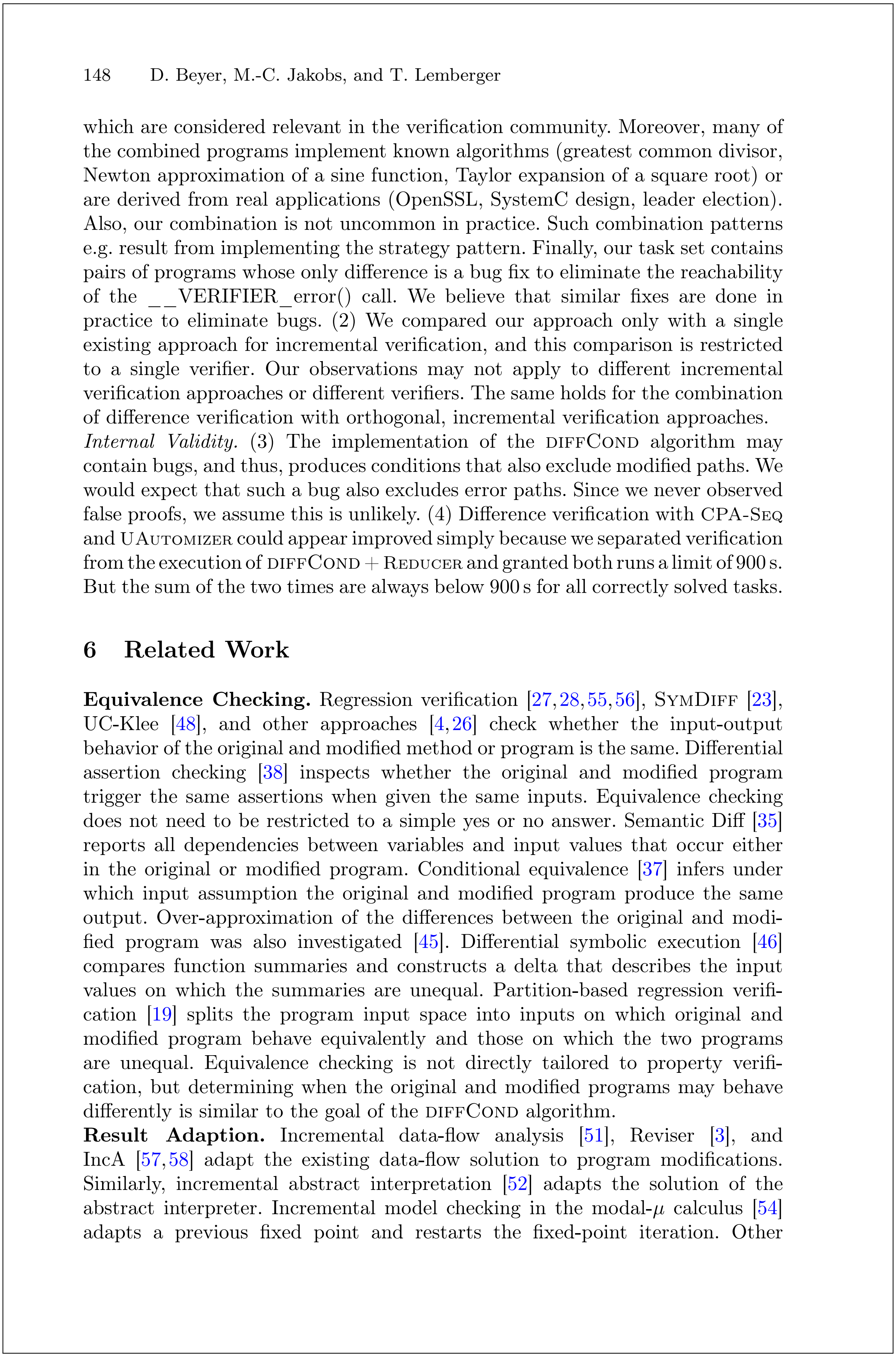
Fig. 8. CPU time (in s) of (a) full difference-verification runs and the time spent for the two diff. components diffCond + reducer, (b) Predicate with precision reuse (Predicate$^{\circlearrowright}$) vs. Predicate with difference verification (Predicate$^{\Delta}$), and (c) Predicate$^{\Delta}$ vs. Predicate$^{\Delta}$ with precision reuse (Predicate$^{\Delta\circlearrowright}$)

Claim 5 (Difference verification with conditions complementary). To show that difference verification with conditions complements existing incremental verification, we need to compare difference verification with conditions against an existing incremental approach. Looking at existing approaches that are (1) available as replication artifact and (b) able to run on verification tasks from sv-benchmarks, we identified two: both based on precision reuse, one implemented in CPAchecker [16] and one in Ultimate [49]. We use the one in CPAchecker. Figure 8b shows the CPU time of precision reuse with Predicate, called Predicate$^{\circlearrowright}$ (x-axis) against our difference verification with Predicate, called Predicate$^{\Delta}$ (y-axis). Many tasks are solved efficiently by both techniques (large cluster in lower left). For the remaining hard tasks, difference verification is often faster than precision reuse, or precision reuse cannot even solve the task (points below the diagonal and on right edge). This shows that difference verification with conditions can improve on precision reuse for a significant number of tasks. It can thus complement existing incremental techniques.

Claim 6 (Combinations sometimes beneficial). We combined difference verification with conditions with precision reuse, called Predicate$^{\Delta\circlearrowright}$. Figure 8c shows that this combination rarely becomes faster than difference verification Predicate$^{\Delta}$ alone. In the worst case, the combination even slows down because precision reuse tracks previously used predicates from the beginning while difference verification would only detect the necessary ones lazily. This more precise abstraction leads to more, sometimes unnecessary computations. Nevertheless, the combination can solve 29 tasks that neither Predicate, its difference verifier, nor precision reuse can solve alone. Thus, while a combination of the two incremental techniques is not beneficial in general, it can be.

5.3 Threats to Validity

External Validity. (1) Our benchmark tasks might not represent real program changes, and thus, our results might not transfer to reality. However, we built our tasks from a well-established collection of software-verification problems,



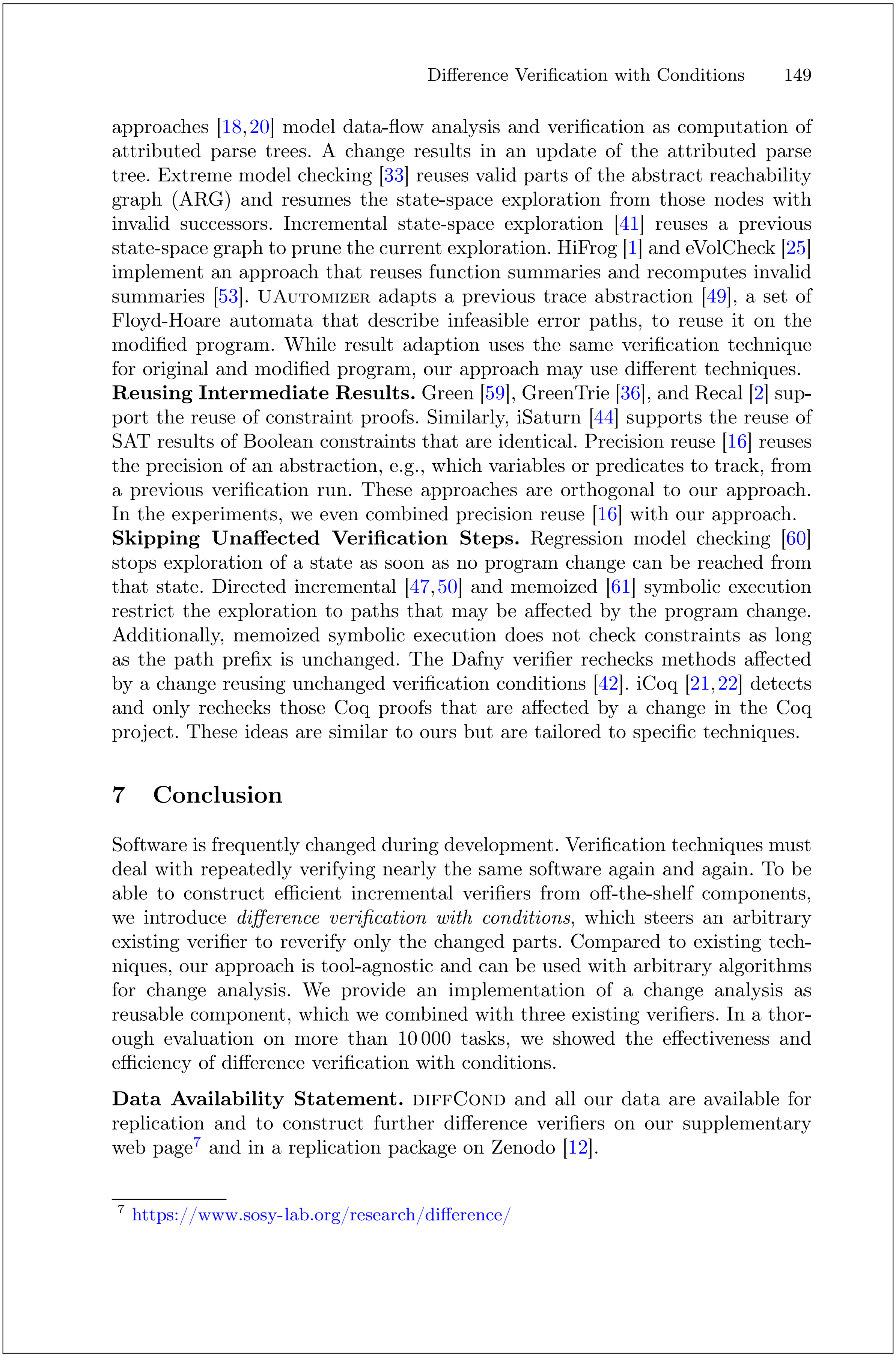
which are considered relevant in the verification community. Moreover, many of the combined programs implement known algorithms (greatest common divisor, Newton approximation of a sine function, Taylor expansion of a square root) or are derived from real applications (OpenSSL, SystemC design, leader election). Also, our combination is not uncommon in practice. Such combination patterns e.g. result from implementing the strategy pattern. Finally, our task set contains pairs of programs whose only difference is a bug fix to eliminate the reachability of the __VERIFIER_error() call. We believe that similar fixes are done in practice to eliminate bugs. (2) We compared our approach only with a single existing approach for incremental verification, and this comparison is restricted to a single verifier. Our observations may not apply to different incremental verification approaches or different verifiers. The same holds for the combination of difference verification with orthogonal, incremental verification approaches.

Internal Validity. (3) The implementation of the diffCond algorithm may contain bugs, and thus, produces conditions that also exclude modified paths. We would expect that such a bug also excludes error paths. Since we never observed false proofs, we assume this is unlikely. (4) Difference verification with CPA-Seq and UAutomizer could appear improved simply because we separated verification from the execution of diffCond + Reducer and granted both runs a limit of 900 s. But the sum of the two times are always below 900 s for all correctly solved tasks.

6 Related Work

Equivalence Checking. Regression verification [27,28,55,56], SymDiff [23], UC-Klee [48], and other approaches [4,26] check whether the input-output behavior of the original and modified method or program is the same. Differential assertion checking [38] inspects whether the original and modified program trigger the same assertions when given the same inputs. Equivalence checking does not need to be restricted to a simple yes or no answer. Semantic Diff [35] reports all dependencies between variables and input values that occur either in the original or modified program. Conditional equivalence [37] infers under which input assumption the original and modified program produce the same output. Over-approximation of the differences between the original and modified program was also investigated [45]. Differential symbolic execution [46] compares function summaries and constructs a delta that describes the input values on which the summaries are unequal. Partition-based regression verification [19] splits the program input space into inputs on which original and modified program behave equivalently and those on which the two programs are unequal. Equivalence checking is not directly tailored to property verification, but determining when the original and modified programs may behave differently is similar to the goal of the diffCond algorithm.

Result Adaption. Incremental data-flow analysis [51], Reviser [3], and IncA [57,58] adapt the existing data-flow solution to program modifications. Similarly, incremental abstract interpretation [52] adapts the solution of the abstract interpreter. Incremental model checking in the modal-μ calculus [54] adapts a previous fixed point and restarts the fixed-point iteration. Other



approaches [18,20] model data-flow analysis and verification as computation of attributed parse trees. A change results in an update of the attributed parse tree. Extreme model checking [33] reuses valid parts of the abstract reachability graph (ARG) and resumes the state-space exploration from those nodes with invalid successors. Incremental state-space exploration [41] reuses a previous state-space graph to prune the current exploration. HiFrog [1] and eVolCheck [25] implement an approach that reuses function summaries and recomputes invalid summaries [53]. UAutomizer adapts a previous trace abstraction [49], a set of Floyd-Hoare automata that describe infeasible error paths, to reuse it on the modified program. While result adaption uses the same verification technique for original and modified program, our approach may use different techniques.

Reusing Intermediate Results. Green [59], GreenTrie [36], and Recal [2] support the reuse of constraint proofs. Similarly, iSaturn [44] supports the reuse of SAT results of Boolean constraints that are identical. Precision reuse [16] reuses the precision of an abstraction, e.g., which variables or predicates to track, from a previous verification run. These approaches are orthogonal to our approach. In the experiments, we even combined precision reuse [16] with our approach.

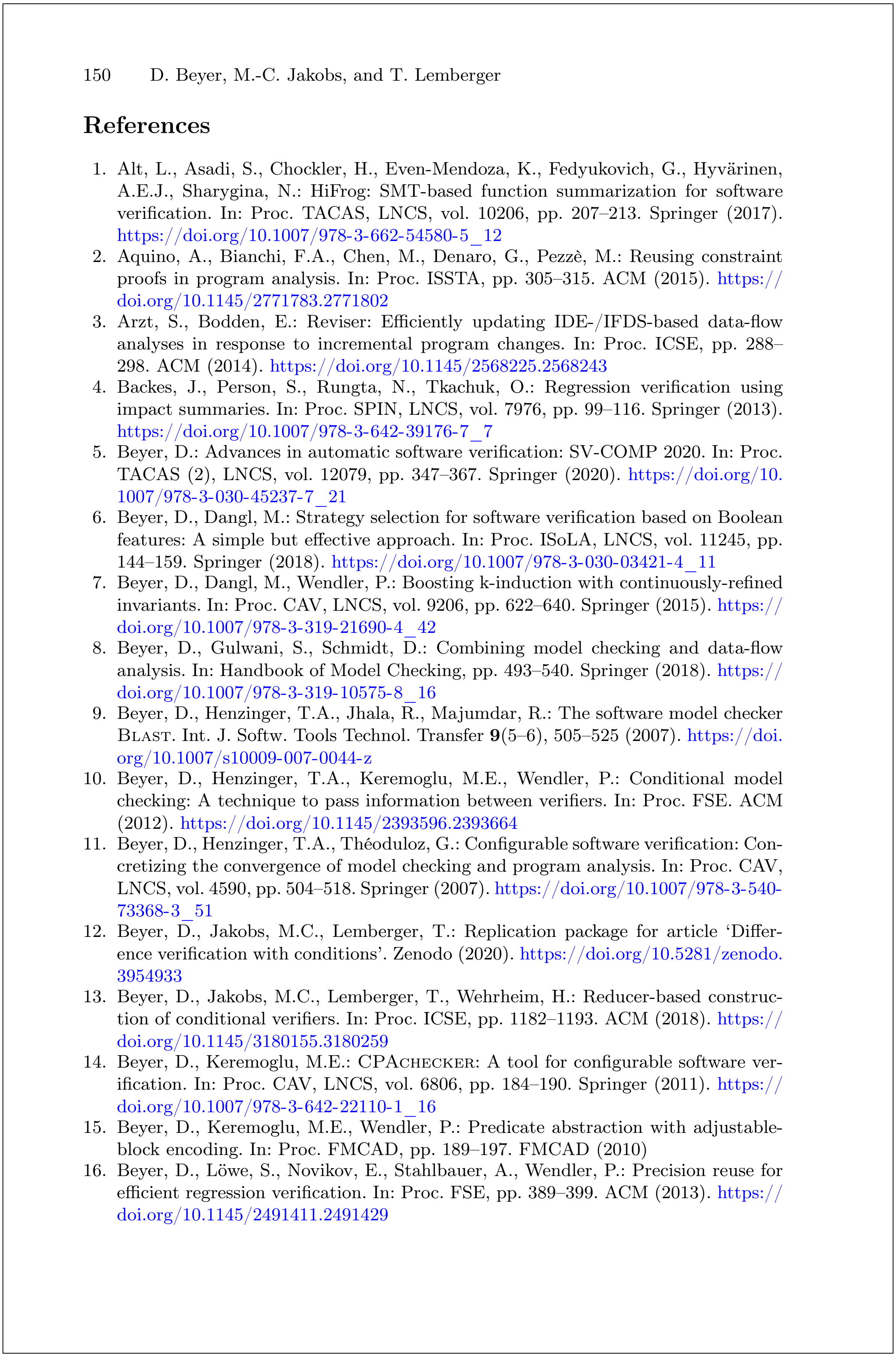
Skipping Unaffected Verification Steps. Regression model checking [60] stops exploration of a state as soon as no program change can be reached from that state. Directed incremental [47,50] and memoized [61] symbolic execution restrict the exploration to paths that may be affected by the program change. Additionally, memoized symbolic execution does not check constraints as long as the path prefix is unchanged. The Dafny verifier rechecks methods affected by a change reusing unchanged verification conditions [42]. iCoq [21,22] detects and only rechecks those Coq proofs that are affected by a change in the Coq project. These ideas are similar to ours but are tailored to specific techniques.

7 Conclusion

Software is frequently changed during development. Verification techniques must deal with repeatedly verifying nearly the same software again and again. To be able to construct efficient incremental verifiers from off-the-shelf components, we introduce *difference verification with conditions*, which steers an arbitrary existing verifier to reverify only the changed parts. Compared to existing techniques, our approach is tool-agnostic and can be used with arbitrary algorithms for change analysis. We provide an implementation of a change analysis as reusable component, which we combined with three existing verifiers. In a thorough evaluation on more than 10 000 tasks, we showed the effectiveness and efficiency of difference verification with conditions.

Data Availability Statement. diffCond and all our data are available for replication and to construct further difference verifiers on our supplementary web page[7] and in a replication package on Zenodo [12].

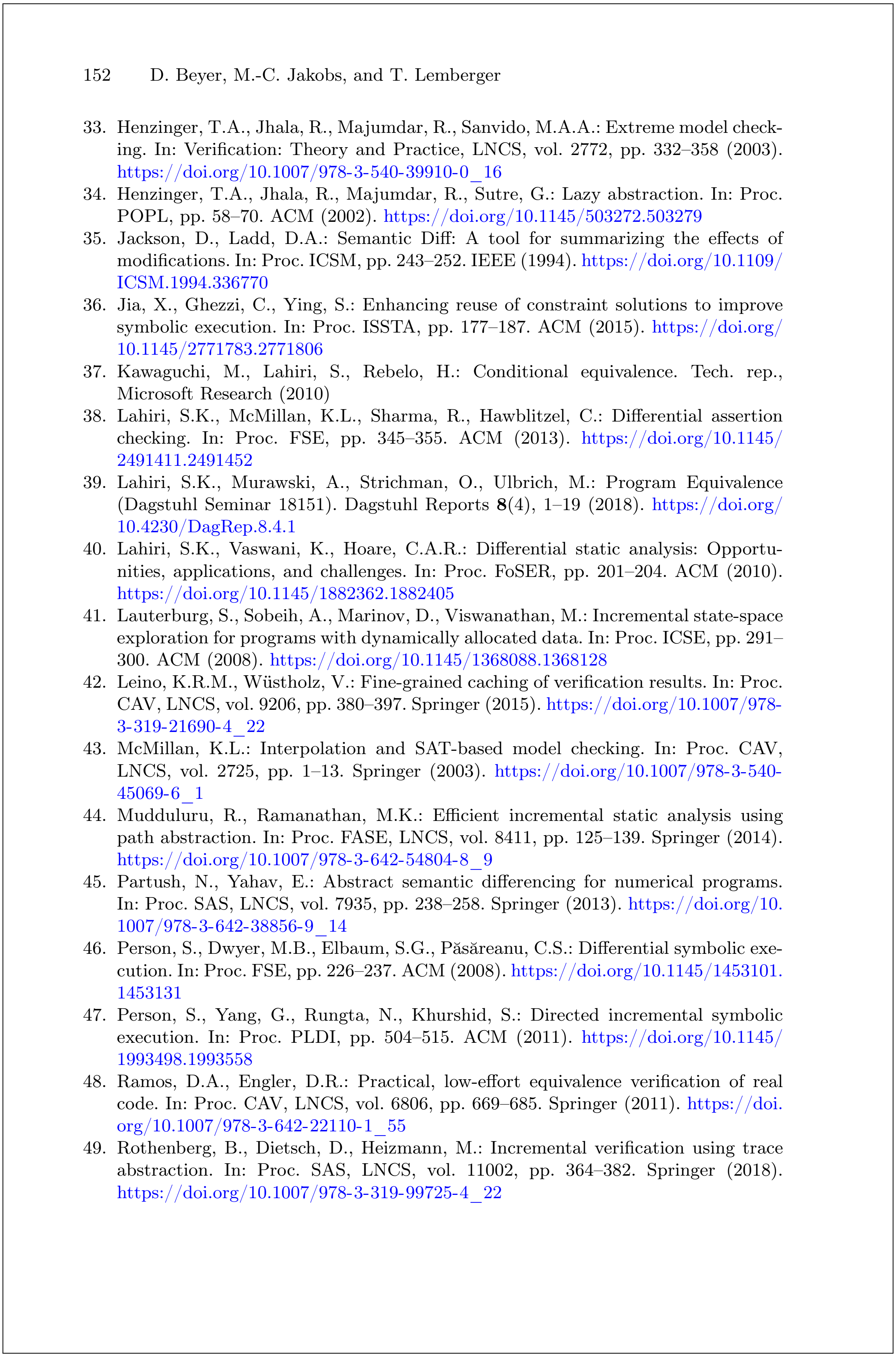
[7] https://www.sosy-lab.org/research/difference/



References

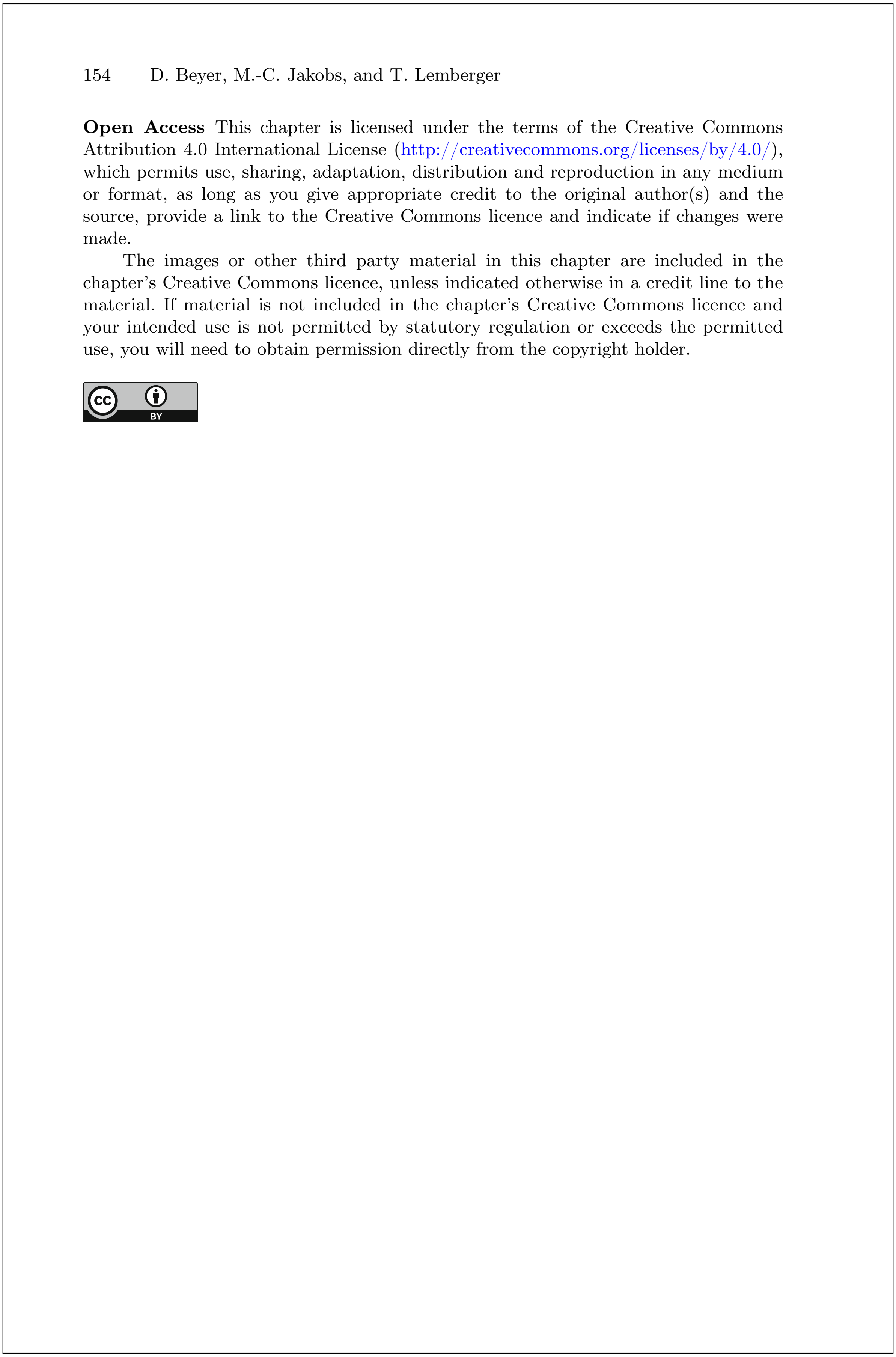
1. Alt, L., Asadi, S., Chockler, H., Even-Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: HiFrog: SMT-based function summarization for software verification. In: Proc. TACAS, LNCS, vol. 10206, pp. 207–213. Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_12
2. Aquino, A., Bianchi, F.A., Chen, M., Denaro, G., Pezzè, M.: Reusing constraint proofs in program analysis. In: Proc. ISSTA, pp. 305–315. ACM (2015). https://doi.org/10.1145/2771783.2771802
3. Arzt, S., Bodden, E.: Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In: Proc. ICSE, pp. 288–298. ACM (2014). https://doi.org/10.1145/2568225.2568243
4. Backes, J., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: Proc. SPIN, LNCS, vol. 7976, pp. 99–116. Springer (2013). https://doi.org/10.1007/978-3-642-39176-7_7
5. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2), LNCS, vol. 12079, pp. 347–367. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
6. Beyer, D., Dangl, M.: Strategy selection for software verification based on Boolean features: A simple but effective approach. In: Proc. ISoLA, LNCS, vol. 11245, pp. 144–159. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11
7. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV, LNCS, vol. 9206, pp. 622–640. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
8. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
9. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer **9**(5–6), 505–525 (2007). https://doi.org/10.1007/s10009-007-0044-z
10. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV, LNCS, vol. 4590, pp. 504–518. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
12. Beyer, D., Jakobs, M.C., Lemberger, T.: Replication package for article ‘Difference verification with conditions’. Zenodo (2020). https://doi.org/10.5281/zenodo.3954933
13. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE, pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
14. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
15. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
16. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE, pp. 389–399. ACM (2013). https://doi.org/10.1145/2491411.2491429

17. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proc. SPIN, LNCS, vol. 9232, pp. 160–178. Springer (2015). https://doi.org/10.1007/978-3-319-23404-5_12
18. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: Syntactic-semantic incrementality for agile verification. SCICO **97**, 47–54 (2015). https://doi.org/10.1016/j.scico.2013.11.026
19. Böhme, M., Oliveira, B.C.d.S., Roychoudhury, A.: Partition-based regression verification. In: Proc. ICSE, pp. 302–311. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606576
20. Carroll, M.D., Ryder, B.G.: Incremental data-flow analysis via dominator and attribute updates. In: Proc. POPL, pp. 274–284. ACM (1988). https://doi.org/10.1145/73560.73584
21. Çelik, A., Palmskog, K., Gligoric, M.: iCoq: Regression proof selection for large-scale verification projects. In: Proc. ASE, pp. 171–182. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115630
22. Çelik, A., Palmskog, K., Gligoric, M.: A regression proof selection tool for Coq. In: Proc. ICSE (Companion Volume), pp. 117–120. ACM (2018). https://doi.org/10.1145/3183440.3183493
23. Chaki, S., Gurfinkel, A., Strichman, O.: Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation). FMSD **47**(3), 287–301 (2015). https://doi.org/10.1007/s10703-015-0237-0
24. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
25. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental upgrade checker for C. In: Proc. TACAS, LNCS, vol. 7795, pp. 292–307. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_21
26. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proc. ASE, pp. 349–360. ACM (2014). https://doi.org/10.1145/2642937.2642987
27. Godlin, B., Strichman, O.: Regression verification. In: Proc. DAC, pp. 466–471. ACM (2009). https://doi.org/10.1145/1629911.1630034
28. Godlin, B., Strichman, O.: Regression verification: Proving the equivalence of similar programs. Softw. Test. Verif. Reliab. **23**(3), 241–258 (2013). https://doi.org/10.1002/stvr.1472
29. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2), LNCS, vol. 10806, pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
30. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS, LNCS, vol. 5673, pp. 69–85. Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
31. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV, LNCS, vol. 8044, pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
32. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL, pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021



33. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Verification: Theory and Practice, LNCS, vol. 2772, pp. 332–358 (2003). https://doi.org/10.1007/978-3-540-39910-0_16
34. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
35. Jackson, D., Ladd, D.A.: Semantic Diff: A tool for summarizing the effects of modifications. In: Proc. ICSM, pp. 243–252. IEEE (1994). https://doi.org/10.1109/ICSM.1994.336770
36. Jia, X., Ghezzi, C., Ying, S.: Enhancing reuse of constraint solutions to improve symbolic execution. In: Proc. ISSTA, pp. 177–187. ACM (2015). https://doi.org/10.1145/2771783.2771806
37. Kawaguchi, M., Lahiri, S., Rebelo, H.: Conditional equivalence. Tech. rep., Microsoft Research (2010)
38. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proc. FSE, pp. 345–355. ACM (2013). https://doi.org/10.1145/2491411.2491452
39. Lahiri, S.K., Murawski, A., Strichman, O., Ulbrich, M.: Program Equivalence (Dagstuhl Seminar 18151). Dagstuhl Reports **8**(4), 1–19 (2018). https://doi.org/10.4230/DagRep.8.4.1
40. Lahiri, S.K., Vaswani, K., Hoare, C.A.R.: Differential static analysis: Opportunities, applications, and challenges. In: Proc. FoSER, pp. 201–204. ACM (2010). https://doi.org/10.1145/1882362.1882405
41. Lauterburg, S., Sobeih, A., Marinov, D., Viswanathan, M.: Incremental state-space exploration for programs with dynamically allocated data. In: Proc. ICSE, pp. 291–300. ACM (2008). https://doi.org/10.1145/1368088.1368128
42. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: Proc. CAV, LNCS, vol. 9206, pp. 380–397. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_22
43. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV, LNCS, vol. 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
44. Mudduluru, R., Ramanathan, M.K.: Efficient incremental static analysis using path abstraction. In: Proc. FASE, LNCS, vol. 8411, pp. 125–139. Springer (2014). https://doi.org/10.1007/978-3-642-54804-8_9
45. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: Proc. SAS, LNCS, vol. 7935, pp. 238–258. Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_14
46. Person, S., Dwyer, M.B., Elbaum, S.G., Păsăreanu, C.S.: Differential symbolic execution. In: Proc. FSE, pp. 226–237. ACM (2008). https://doi.org/10.1145/1453101.1453131
47. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Proc. PLDI, pp. 504–515. ACM (2011). https://doi.org/10.1145/1993498.1993558
48. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proc. CAV, LNCS, vol. 6806, pp. 669–685. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_55
49. Rothenberg, B., Dietsch, D., Heizmann, M.: Incremental verification using trace abstraction. In: Proc. SAS, LNCS, vol. 11002, pp. 364–382. Springer (2018). https://doi.org/10.1007/978-3-319-99725-4_22

50. Rungta, N., Person, S., Branchaud, J.: A change impact analysis to characterize evolving program behaviors. In: Proc. ICSM, pp. 109–118. IEEE (2012). https://doi.org/10.1109/ICSM.2012.6405261
51. Ryder, B.G.: Incremental data-flow analysis. In: Proc. POPL, pp. 167–176. ACM (1983). https://doi.org/10.1145/567067.567084
52. Seidl, H., Erhard, J., Vogler, R.: Incremental abstract interpretation. In: From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement, LNCS, vol. 12065, pp. 132–148. Springer (2020). https://doi.org/10.1007/978-3-030-41103-9_5
53. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: Proc. FMCAD, pp. 114–121. FMCAD Inc. (2012)
54. Sokolsky, O.V., Smolka, S.A.: Incremental model checking in the modal mu-calculus. In: Proc. CAV, LNCS, vol. 818, pp. 351–363. Springer (1994). https://doi.org/10.1007/3-540-58179-0_67
55. Strichman, O., Godlin, B.: Regression verification – a practical way to verify programs. In: Proc. VSTTE, LNCS, vol. 4171, pp. 496–501. Springer (2008). https://doi.org/10.1007/978-3-540-69149-5_54
56. Strichman, O., Veitsman, M.: Regression verification for unbalanced recursive functions. In: Proc. FM, LNCS, vol. 9995, pp. 645–658 (2016). https://doi.org/10.1007/978-3-319-48989-6_39
57. Szabó, T., Bergmann, G., Erdweg, S., Voelter, M.: Incrementalizing lattice-based program analyses in Datalog. PACMPL **2**(OOPSLA), 139:1–139:29 (2018). https://doi.org/10.1145/3276509
58. Szabó, T., Erdweg, S., Voelter, M.: IncA: A DSL for the definition of incremental program analyses. In: Proc. ASE, pp. 320–331. ACM (2016). https://doi.org/10.1145/2970276.2970298
59. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: Reducing, reusing, and recycling constraints in program analysis. In: Proc. FSE, pp. 58:1–58:11. ACM (2012). https://doi.org/10.1145/2393596.2393665
60. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: Proc. ICSM, pp. 115–124. IEEE (2009). https://doi.org/10.1109/ICSM.2009.5306334
61. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: Proc. ISSTA, pp. 144–154. ACM (2012). https://doi.org/10.1145/2338965.2336771
62. Yoo, S., Harman, M.: Regression testing minimization, selection, and prioritization: A survey. STVR **22**(2), 67–120 (2012). https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.430



Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR

Dirk Beyer
dirk.beyer@sosy-lab.org
LMU Munich
Munich, Germany

Jan Haltermann*
jan.haltermann@uol.de
University of Oldenburg
Oldenburg, Germany

Thomas Lemberger*
thomas.lemberger@sosy.ifi.lmu.de
LMU Munich
Munich, Germany

Heike Wehrheim
heike.wehrheim@uol.de
University of Oldenburg
Oldenburg, Germany

ABSTRACT

Techniques for software verification are typically realized as cohesive units of software with tightly coupled components. This makes it difficult to re-use components, and the potential for workload distribution is limited. Innovations in software verification might find their way into practice faster if provided in smaller, more specialized components.

In this paper, we propose to strictly decompose software verification: the verification task is split into independent subtasks, implemented by only loosely coupled components communicating via clearly defined interfaces. We apply this decomposition concept to one of the most frequently employed techniques in software verification: counterexample-guided abstraction refinement (CEGAR). CEGAR is a technique to iteratively compute an abstract model of the system. We develop a decomposition of CEGAR into independent components with clearly defined interfaces that are based on existing, standardized exchange formats. Its realization *component-based CEGAR (C-CEGAR)* concerns the three core tasks of CEGAR: abstract-model exploration, feasibility check, and precision refinement. We experimentally show that — despite the necessity of exchanging complex data via interfaces — the efficiency thereby only reduces by a small constant factor while the precision in solving verification tasks even increases. We furthermore illustrate the advantages of C-CEGAR by experimenting with different implementations of components, thereby further increasing the overall effectiveness and testing that substitution of components works well.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; **Abstraction, modeling and modularity**; • **Theory of computation** → **Logic and verification**.

[*] Both authors contributed equally to this research.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510064

KEYWORDS

Software engineering, Software verification, Abstraction refinement, CEGAR, Decomposition, Cooperative verification

ACM Reference Format:
Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. 2022. Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510064

1 INTRODUCTION

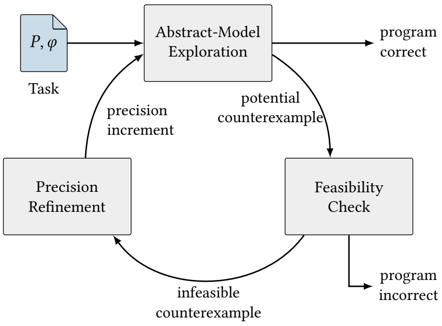


Figure 1: Workflow of classic CEGAR

Over the past decades, software verification has emerged as an area with continuous research innovations and also with increasing tool development. Competitions on software verification (SV-COMP [15], VerifyThis [67]) showcase and conserve the rapid process of tool building and application, and have also observed an interest in standardization of verification artifacts (e.g., of verification witnesses). The general task of automatic verification tools is to compute a proof or counterexample for specified requirements.

Today, the majority of existing verification tools, whether configurable or not, are strongly cohesive software units. Though software verification as a task clearly consists of individual subtasks, verifiers are typically made up of tightly coupled, stateful components

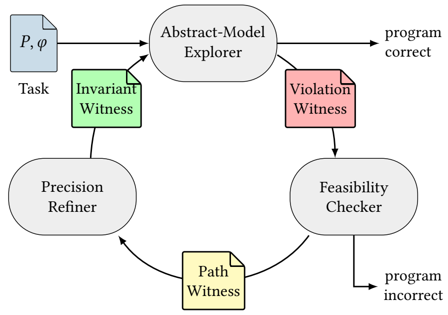


Figure 2: Workflow of component-based CEGAR

that operate on shared data structures. This architecture complicates reuse of components, impacts scalability (e.g., parallelization) and hampers exchange and integration of new components. In consequence, it often requires major implementation effort to integrate innovations in verification technology into existing tools, or is even prohibitive because the strong cohesion between existing components can not be broken easily.

To avoid this issue, we propose to employ *decomposition* concepts in the construction of verifiers. Instead of having all components integrated into a single tool, we opt for *cooperative* verification [41] where independent executable units cooperate on a verification task. Every such unit is only responsible for one well-defined subtask, and the units communicate via clearly defined interfaces.

To investigate the feasibility of such ideas, we have realized this strict decomposition into components for one of the most frequently employed techniques in software verification, *counterexample-guided abstraction refinement* (CEGAR) [53, 54]. CEGAR is a technique for automatically finding an abstract model of the software to be verified which is *as abstract as possible*, but *as precise as necessary* to successfully construct a proof of correctness or a refutation. Many tools for software verification include this CEGAR principle (e.g., [2, 4, 25, 34, 45, 47, 71, 76, 81, 83, 94, 96, 101–103]). CEGAR is also successfully employed in other areas, like probabilistic or timed-automata model checking [79, 80]. CEGAR readily lends itself to a decomposition which we realize here as *component-based CEGAR (C-CEGAR)*. Figure 1 first of all illustrates the iterative procedure of classic CEGAR: For a given level of abstraction, the exploration of an abstract model of the software (top) either proves the program correct and terminates the procedure, or finds a potential counterexample. The feasibility check (right) analyzes the counterexample. It either proves the counterexample feasible and terminates the procedure, or passes an infeasible counterexample to the next phase. The precision refinement (left) analyzes the infeasible counterexample and extracts from it a precision increment refining the abstract model which the abstract-model exploration employs in the next iteration. This cycle continues until either a correctness or a violation proof is found.

But while this general concept of CEGAR has overall proven successful (witnessed by CEGAR-based tools scoring high at SV-COMP), research into specialized techniques for the three subtasks is still ongoing. This can best be illustrated by proposals of and discussions on precision refiners [37, 38, 65, 72]. Precision refinement techniques rely on heuristics, and hence their effectiveness can only be evaluated through experiments. Due to the tight coupling of components in verifiers, new precision refiners can however neither be evaluated in isolation nor can they be integrated into existing tools without reimplementation. The past has thus unfortunately already seen multiple reimplementations of precision refiners: A vast amount of tools [4, 25, 34, 45, 47, 71, 76, 83, 96, 101] contain implementations of a refiner based on Craig interpolation and at least three tools [34, 72, 75] contain (re-)implementations of so called Newton refinement.

C-CEGAR overcomes these disadvantages of classic CEGAR by a consequent decomposition, implementing each of the three conceptual units as a stand-alone component and defining clear-cut interfaces between components. Figure 2 illustrates the workflow of C-CEGAR. For the interfaces, we employ existing standards for verification artifacts, namely violation, path, and invariant *witnesses* [20, 21], but also new formats. Witnesses are already produced by many verifiers, which allows us to partially reuse tools.

We have implemented C-CEGAR as a particular form of cooperative verification, and implemented it using the framework CoVeriTeam [33]. With this implementation at hand, we have then investigated the effects of decomposition into components on the overall effectiveness and efficiency. Our experiments show that while efficiency is slightly impacted by the necessity of data exchange via external interfaces, the overall effectiveness can even be increased. We have moreover performed the now possible independent evaluation of precision refiners, comparing Craig and Newton refinement.

Novelty. We provide the following contributions:

- We develop the concept of C-CEGAR as a composition of three independent (software) units with clearly defined interfaces based on verification witnesses and invariant maps.
- We implement C-CEGAR for C programs using the framework CoVeriTeam.
- We show the feasibility and effectiveness of C-CEGAR through a **sound** experimental evaluation on an extensive benchmark set with 8 347 verification tasks (written in C). We use 3 off-the-shelf verification tools for the three base units.
- We experimentally demonstrate that C-CEGAR makes it possible to independently evaluate components like precision refiners, which was not possible before.
- Our results are **verifiable**: all data and software are publicly available for inspection and reproduction (Sect. 6).

Significance and Potential Impact. Evaluations in the research literature and competitions show that different approaches have different strengths to solve the problem of software verification. It is therefore imperative to leverage the possibility of substituting components by alternatives, instead of (re-)implementing whole tools. Exchangeability is a key feature in component-based design and our approach can lead to components that are tuned to excel in their specific task. SAT and SMT solvers are a success story because there already is such a clearly defined interface (SAT queries, SMTlib exchange format): many applications build on SAT and SMT solvers as components and many tools implement these component interfaces [8, 9, 11, 82].

Software verification needs to be integrated into the continuous-integration process [50], and therefore, it is important to reduce its response time. Most of the currently available software verifiers are not constructed in a way that supports massively parallel execution, but the decomposition of verification techniques into stand-alone components would enable this. With C-CEGAR, we try to improve the state of the art in this respect. We see our work as a catalyst for further research in the domain of CEGAR and as a first step towards a microservice architecture for software verification.

1.1 Related Work

There is a large body of literature on decomposition, interfaces, and cooperative verification. We restrict ourselves here to provide a few pointers to literature that directly inspired our work.

Compositionality and Decomposition. Decomposition is a central general problem-solving approach in computer science, and in particular in software engineering [63, 64]. The goal is to "divide and conquer", that is, split the problem into "easier-to-solve" sub-problems and solve them as independently as possible. Compositionality means that a system can be composed from components.

Cooperative Verification. The electronic tools integration platform (ETI) [86–88, 99] was an effort to collect and conserve tools from the formal-methods community. Wide (and public) availability of tools is the precondition to any kind of cooperation. The evidential tool bus [58, 59, 97] arose also in the formal-methods community, and tries to integrate tools that cooperate, in particular, to compose assurance claims. Conditional model checking (CMC) [26] is an approach in which several tools exchange information about the progress of the verification. CMC introduced a condition as artifact that describes which parts of the system are successfully verified so far. Conditional testing [36] applies the same idea to software testing. Sets of test goals are used as artifact to describe what has been tested so far. Conditions are also used to test what could not be verified [51, 60]. CoDiDroid [91] is a broker that delegates queries to the tools that are best suited to answer them. This way, several tools cooperate to achieve the goal.

Composition in Software Verification. There are several approaches to compose new tools from existing binary components. Reducers [31] can be composed from off-the-shelf verifiers to construct conditional verifiers and the artifact that is passed from the reducer to the verifier is a residual program. MetaVal [40] is an approach to construct a witness-based result validator from a program transformer and an off-the-shelf verifier. If the specification is large, it could be promising to decompose the specification [6].

Interfaces. Components are connected via interfaces. The interface specifies what the outside should know about the component and what types of data (or, more general, artifacts) are expected as input and output. Signatures of functions are often used in programming languages to document how a function can be used, and abstract classes (in Java: interfaces) are used to document a cohesive component or subsystem by a set of functions with their signatures which describe the service that the component or subsystem delivers. Behavioral interfaces were found to be useful for concurrent systems [61], for timed systems [62], for resources [49], for web services [16], and for program APIs and their behavior [28, 32, 77].

Verification Artifacts and Exchange Formats. Artifacts and formats that are relevant for cooperative verification were discussed recently [41]. To give some examples, artifacts for cooperative verification can be (a) programs (exchange format C: [3]), (b) specifications (exchange format: [12, 92]), (c) results (exchange format: [20, 21, 48]), and (d) conditions (exchange format: [10, 11, 26]).

Libraries and Components. Many verification approaches are based on formulas in a certain logic, and theorem provers [98] or SMT solvers [11] are used to reason about the programs or systems. SMT solvers support a standard exchange format [10], and there are even API frameworks [46, 55, 68, 84, 85] that make SMT-solvers exchangeable. There was already an idea to make reachability queries via a defined interface [18], because several verification approaches can be solved with the help of reachability queries, such as termination analysis [56, 93], test-case generation [17, 69], Impact [42, 90], and PDR [19, 43].

CEGAR. The full, concrete system implementation is often complex and abstraction can help to ignore details that are not important for proving correctness or for finding bugs. CEGAR [53, 54] is an approach that can be used to compute an abstraction of the system. There are many verification tools that use CEGAR as a component, for example, the most recent SV-COMP report mentions the following: Brick, CPA-BAM-BnB [101], CPALockator [5], CPAchecker [34], Gazer-Theta [1], JayHorn [83], PeSCo [95], UAutomizer [76], UKojak [66], UTaipan [71], and VeriAbs [2]. CEGAR is a research topic itself because of its importance [27, 37, 38, 44, 73, 90, 100].

This paper stands on the shoulders of the fine works described above: we use CEGAR, decompose it into components, use verification witnesses as interfaces, and reuse existing components for the construction of the components.

2 BACKGROUND

We start by explaining some basic notations and concepts.

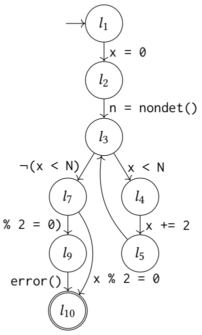
Programs. For a simplified presentation, we assume that each program contains at most one program statement on each source-code line, and that the only variable type is integer ($\mathbb{Z}$). Figures 3a and 3b show two programs. For a program P, we define the set L of all program locations (uniquely identifiable by source-code line), the program counter $pc \in L$, the set X of all program variables, the set Op of all program operations over integer variables and the program states $C = (X \to \mathbb{Z}) \cup (\{pc\} \to L)$. A program state $c \in C$ assigns a value to each program variable, and a line number to pc. A program path $c_0 \xrightarrow{op_0} \dots \xrightarrow{op_{n-1}} c_n$ is a sequence of program states, where c_0 is an initial state with arbitrary value assignments for program variables, op_i is the program statement at program location $c_i(pc)$, and c_{i+1} is a possible successor state of c_i after executing op_i. A *control-flow automaton* (CFA) (L, ℓ_0, G) for a program P consists of the locations L, the program entry ℓ_0 and transitions $G \subseteq L \times Op \times L$, modeling the execution of a statement when moving the program counter from one location to a successor location. When control-flow branches conditionally (e.g., because of an `if-else` or `while`),

```
1 int main() {
2   unsigned int y = 1;
3   while (1) {
4     y = y + 2U * nondet();
5     if (y != 0) {}
6     else
7       error();
8   }
9 }
```

(a) Craig interpolation finds the more meaningful precision ($y \bmod 2 = 1$), Newton finds the equivalent, but more complex precision $1 \leq y + 2 * \lfloor((y * -1 + 1)/2)\rfloor$

```
1  int main(void) {
2    unsigned int x = 0;
3    unsigned short N = nondet();
4    while (x < N) {
5      x += 2;
6    }
7    if (x % 2 == 0) {}
8    else
9      error();
10 }
```

(b) Newton refinement finds the more meaningful precision $x \leq 2 * (x/2)$, Craig interpolation enumerates all valid assignments for x explicitly



(c) CFA for program Fig. 3b

Figure 3: Code examples for Craig interpolation and Newton refinement

the two corresponding edges of a CFA are labeled with the condition (e.g., for the if-branch) and the negated condition (e.g., for the else-branch). Figure 3c gives the CFA of the program in Fig. 3b.

Software verification aims at analyzing the correctness of software. In this work, we suppose the verifier to check C programs for the reachability of calls to the specific error function `error`, which represents a specification violation[1]. A program P is *correct* if there is no program path that contains the statement op_i = `error()`. A state condition ϕ is a logical expression over program variables (e.g., $y = 1$), used to express state-space restrictions. A program state c fulfills a state condition ϕ when $c \models \phi$. We define the type Φ of state conditions.

Witnesses. The interfaces in our component-based CEGAR approach all come in the form of *witnesses.* An *invariant witness* describes a set of potential invariants for a program, using the formal definition of the common exchange format of correctness witness [20][2]. Intuitively, an invariant witness automaton is a CFA equipped with invariants, explaining why a property is not violated on a path or in a program. An example of an invariant witness is given in Fig. 4. More formally, the invariant witness automaton consists of a set of states Q, an initial state q_0 and a transfer relation δ. A state $q \in Q$ may summarize several concrete states of a CFA. In the example, the state q_2 represents the CFA node l_3, q_3 summarizes l_4 and l_5 and q_4 summarizes l_7, l_9 and l_{10}. In addition, states can contain invariants, e.g. state q_2 contains the invariant $x \leq 2 * (x/2)$. A transition between two states is labeled with the line number of a program location. If the location is a branch or a loop-head, the transition is in addition either labeled with `cond-true` or `cond-false`, indicating whether the condition is assumed to be true or false. Each state has an additional self-loop labeled o/w (*otherwise*) that can be taken if no other transition is applicable (when a state summarizes several CFA nodes). Thereby, the invariant witness covers all paths present in a CFA.

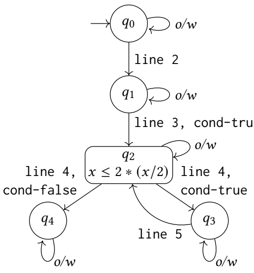


Figure 4: Invariant-witness automaton for Fig. 3b

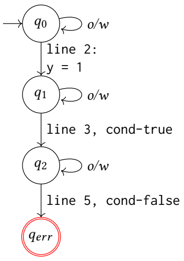


Figure 5: (Invalid) violation-witness automaton for Fig. 3a

A *violation witness* in the common exchange format for witnesses [21] describes a set of program states of which at least one represents a specification violation. These program states are described by a violation witness automaton. A violation witness automaton is similarly defined to invariant witness automata, with three differences: (1) As it only represents a subset of the CFA, it may limit the CFA by not providing a o/w transition for each state, (2) it does not contain invariants and (3) its transitions may, in addition, contain state conditions, to model assumptions on the programs state. Figure 5 contains a violation witness for Fig. 3a, that describes the path of line 1 to line 7. A violation witness is called *valid*, if at least one concrete path in the program matches the described path, otherwise it is *invalid.* As the program in Fig. 3a is correct, the violation witness is invalid.

To increase the confidence in verification results, SV-COMP requires since 2017 [13] that all participating verifiers report a violation witness or correctness witness as part of each verification result.

Precision Refiners. One component of C-CEGAR for which conceptually different techniques exist is the *precision refinement.* In our evaluation, we will illustrate the advantage of C-CEGAR with the possibility of an independent evaluation of precision refiners. Here, we first of all give an example to show that different forms of precision refiners have different benefits. The programs of Figures 3a and 3b (taken from SV-Benchmarks[3]) illustrate the precisions (for a predicate domain) computed by Craig interpolation [57, 78, 89] and Newton refinement [7]. In Fig. 3a, an indefinite while-loop adds a non-deterministically computed even value to program variable y (line 4) and then asserts that $y \neq 0$ (line 5). Because y is initialized with 1 in line 2, y will always stay uneven and thus unequal to 0, even if an overflow occurs. This means that the assertion always holds. Trying to prove this, Craig interpolation (implemented in CPAchecker [34]) computes the predicate $y \bmod 2 = 1$ for the loop head in line 3, which a verifier can use to construct the right level of abstraction for proving the assertion in line 5. Newton refinement

[1] We do not lose generality, as any safety property can be reduced to the call to an arbitrary function.

[2] We use the term invariant witness, as a correctness witness contains correct invariants only, in contrast to the invariant witness.

[3] https://github.com/sosy-lab/sv-benchmarks

(implemented in Ultimate Automizer [65]) computes a predicate with the same meaning, but it is more complex and increases verification overhead: $1 \leq y+2*\lfloor((y*-1+1)/2)\rfloor$[4] In Fig. 3b, a while-loop adds value 2 to program variable x until x is greater than or equal to non-deterministic value N. Afterwards, it asserts that $x \bmod 2 = 0$. Because x is initialized with 0 and N is of type `unsigned short`, the loop can at most add $2 * 65536$ to x. The type of x, `unsigned int`, is large enough to hold this value. Thus, there will be no overflow on x, and x will always be even (or 0). This means that the assertion always holds. Trying to prove this, Craig interpolation creates a large number of predicates that enumerate all possible values for x, i.e., $x = 0$, $x = 2$, $x = 4$, etc. Computing these predicates requires many precision refinements and is costly. In contrast, Newton refinement finds the more helpful predicate, $x \leq 2 * (x/2)$, which encodes $x \bmod 2 = 0$ and hints to a more suited, coarse abstraction. These small examples show that there is no single technique that is optimal for all programs. C-CEGAR is designed to open up the possibility for systematically performing exactly these kind of comparisons.

3 COMPONENT-BASED CEGAR

For a decomposition of CEGAR, we need to identify its individual *components*, and precisely define the *interfaces* between components. Figure 2 depicts the resulting workflow.

3.1 Interfaces of C-CEGAR

The components of CEGAR pass (infeasible) counterexamples and precision increments among each other. We briefly discuss the information passed:

Paths. Both potential and infeasible counterexamples are typically described by (sets of) *program paths*. Program paths are sequences of program locations and program states. An exchange format for paths should allow to describe program paths both concrete and abstract, so that multiple paths can be described and information can be restricted to the important. The exact path information that is exchanged must balance precision and abstraction: A more precise description of program paths avoids imprecision, but may become very large (think about a precise description of many loop unrollings) and lead the precision refiner to produce very specific precision increments. A more abstract description of program paths may guide a precision refiner to produce more generic precision increments (which are often better), but it may also increase imprecision and require more time to analyze. It may be beneficial to not only describe syntactic program paths, but to include information about the program state, like constraints on variable values for reaching a certain program location. This can help the feasibility checker and precision refiner to reconstruct relevant information.

Precision Increment. The precision increment produced for an infeasible counterexample helps the abstract-model exploration to not explore the same infeasible counterexample again, but to prove it infeasible. The concrete type of precision depends on the abstract model explorer, but for communicating precision increments, we propose the use of partial invariants, i.e., invariants that hold for a subset of program paths. From these partial invariants, an abstract model explorer can infer its precision. For example, predicate abstraction can split partial invariants into atoms and create a mapping from program locations to predicates. When exchanging partial invariants, a balance between weak and strong invariants must be found. In addition, most of the time, smaller invariants are easier to parse and reuse than equivalent, but more complex invariants (consider the example invariants of Fig. 3a).

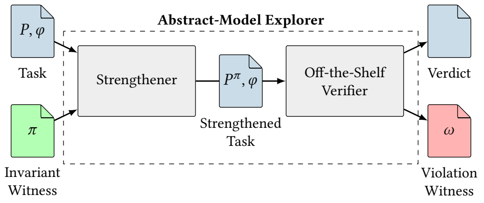


Figure 6: Construction of an abstract-model explorer from an off-the-shelf verifier

The types of these artifacts are arbitrary and information other than the proposed are possible. But to achieve the goals of C-CEGAR, common (and at the best standardized) interfaces are required. To exchange sets of program paths and precision increments, we propose to use the existing exchange formats for verification artifacts [21, 41]:

Violation Witness. We use violation witnesses for describing the (potentially infeasible) counterexample obtained from the abstract-model exploration.

Path Witness. If a violation witness is rejected by the feasibility checker, then it describes an infeasible counterexample path and no valid violation. To signify this change in the meaning of the witness, we call a rejected violation witness *path witness*. A path witness is passed from feasibility checker to precision refiner.

Invariant Witness. Precision increments are described by invariant witnesses which give (partial) invariants in a program, e.g., for invariants associated to loop heads.

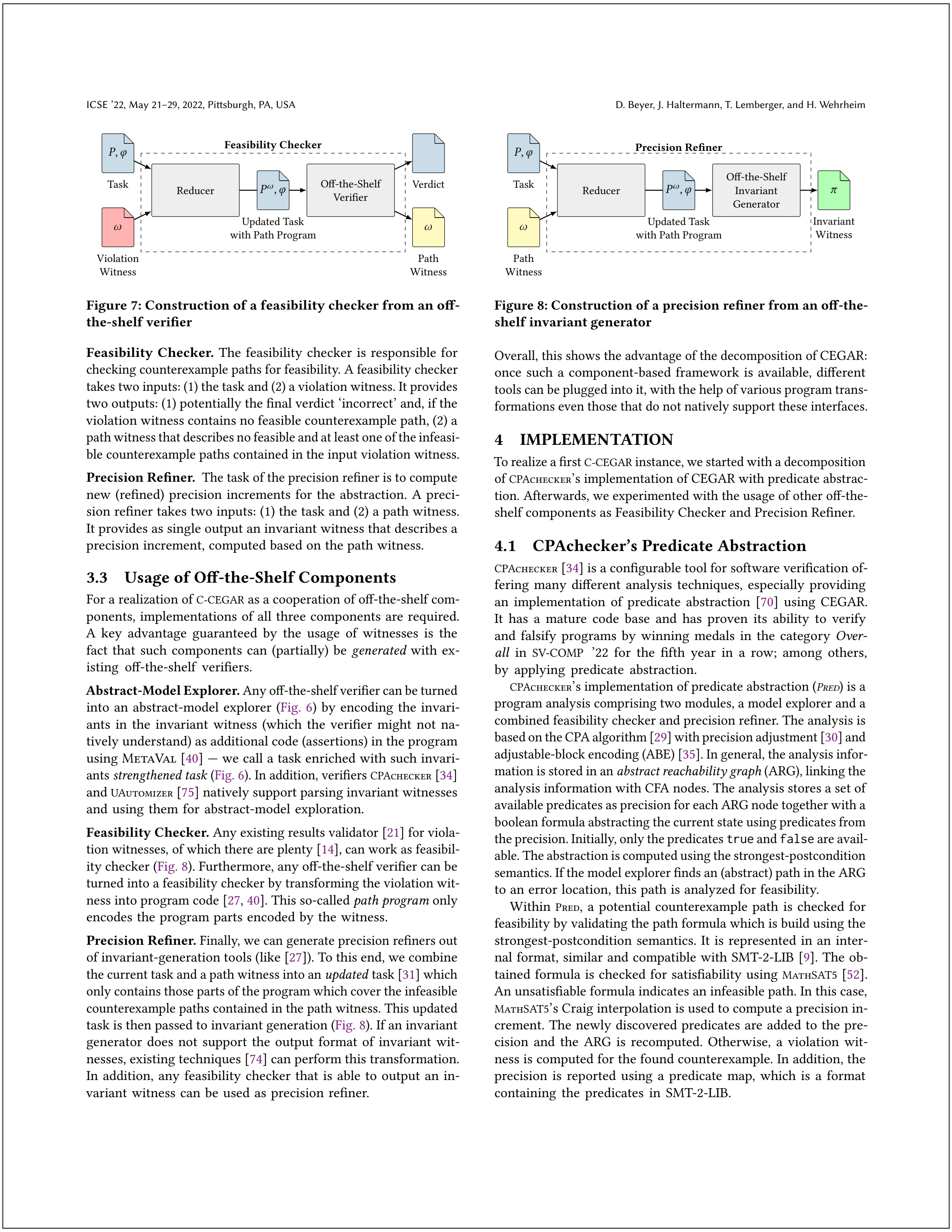
With the existence of a general format for witnesses [21], we thus have tool-independent interfaces.

3.2 Components of C-CEGAR

Next, we describe the three components of C-CEGAR in more detail. The three components use the above interfaces to pass information from one to the next component. Furthermore, the components take input from and provide output to the environment.

Abstract-Model Explorer. C-CEGAR uses an *abstract-model explorer* to compute the abstraction. The abstract-model explorer takes two inputs: (1) the program P under verification and the specification φ (together called *task*), and (2) an invariant witness, and provides two outputs: (1) potentially the final verdict 'correct', and, if the verdict is not 'correct', (2) a violation witness that describes at least one potential counterexample path. The input invariant witness describes the precision increment. The contained (partial) invariants are parsed and used for improving the precision of the abstraction employed during model exploration.

[4] It can be shown that this term is equivalent to $y \bmod 2 = 1$ based on the C datatypes. A detailed reasoning is given on our supplementary webpage, https://www.sosy-lab.org/research/component-based-cegar/.



Feasibility Checker
P, φ
Task
ω
Violation Witness
Reducer
P^ω, φ
Updated Task with Path Program
Off-the-Shelf Verifier
Verdict
ω
Path Witness

Figure 7: Construction of a feasibility checker from an off-the-shelf verifier

Precision Refiner
P, φ
Task
ω
Path Witness
Reducer
P^ω, φ
Updated Task with Path Program
Off-the-Shelf Invariant Generator
π
Invariant Witness

Figure 8: Construction of a precision refiner from an off-the-shelf invariant generator

Feasibility Checker. The feasibility checker is responsible for checking counterexample paths for feasibility. A feasibility checker takes two inputs: (1) the task and (2) a violation witness. It provides two outputs: (1) potentially the final verdict 'incorrect' and, if the violation witness contains no feasible counterexample path, (2) a path witness that describes no feasible and at least one of the infeasible counterexample paths contained in the input violation witness.

Precision Refiner. The task of the precision refiner is to compute new (refined) precision increments for the abstraction. A precision refiner takes two inputs: (1) the task and (2) a path witness. It provides as single output an invariant witness that describes a precision increment, computed based on the path witness.

3.3 Usage of Off-the-Shelf Components

For a realization of C-CEGAR as a cooperation of off-the-shelf components, implementations of all three components are required. A key advantage guaranteed by the usage of witnesses is the fact that such components can (partially) be *generated* with existing off-the-shelf verifiers.

Abstract-Model Explorer. Any off-the-shelf verifier can be turned into an abstract-model explorer (Fig. 6) by encoding the invariants in the invariant witness (which the verifier might not natively understand) as additional code (assertions) in the program using MetaVal [40] — we call a task enriched with such invariants *strengthened task* (Fig. 6). In addition, verifiers CPAchecker [34] and UAutomizer [75] natively support parsing invariant witnesses and using them for abstract-model exploration.

Feasibility Checker. Any existing results validator [21] for violation witnesses, of which there are plenty [14], can work as feasibility checker (Fig. 8). Furthermore, any off-the-shelf verifier can be turned into a feasibility checker by transforming the violation witness into program code [27, 40]. This so-called *path program* only encodes the program parts encoded by the witness.

Precision Refiner. Finally, we can generate precision refiners out of invariant-generation tools (like [27]). To this end, we combine the current task and a path witness into an *updated* task [31] which only contains those parts of the program which cover the infeasible counterexample paths contained in the path witness. This updated task is then passed to invariant generation (Fig. 8). If an invariant generator does not support the output format of invariant witnesses, existing techniques [74] can perform this transformation. In addition, any feasibility checker that is able to output an invariant witness can be used as precision refiner.

Overall, this shows the advantage of the decomposition of CEGAR: once such a component-based framework is available, different tools can be plugged into it, with the help of various program transformations even those that do not natively support these interfaces.

4 IMPLEMENTATION

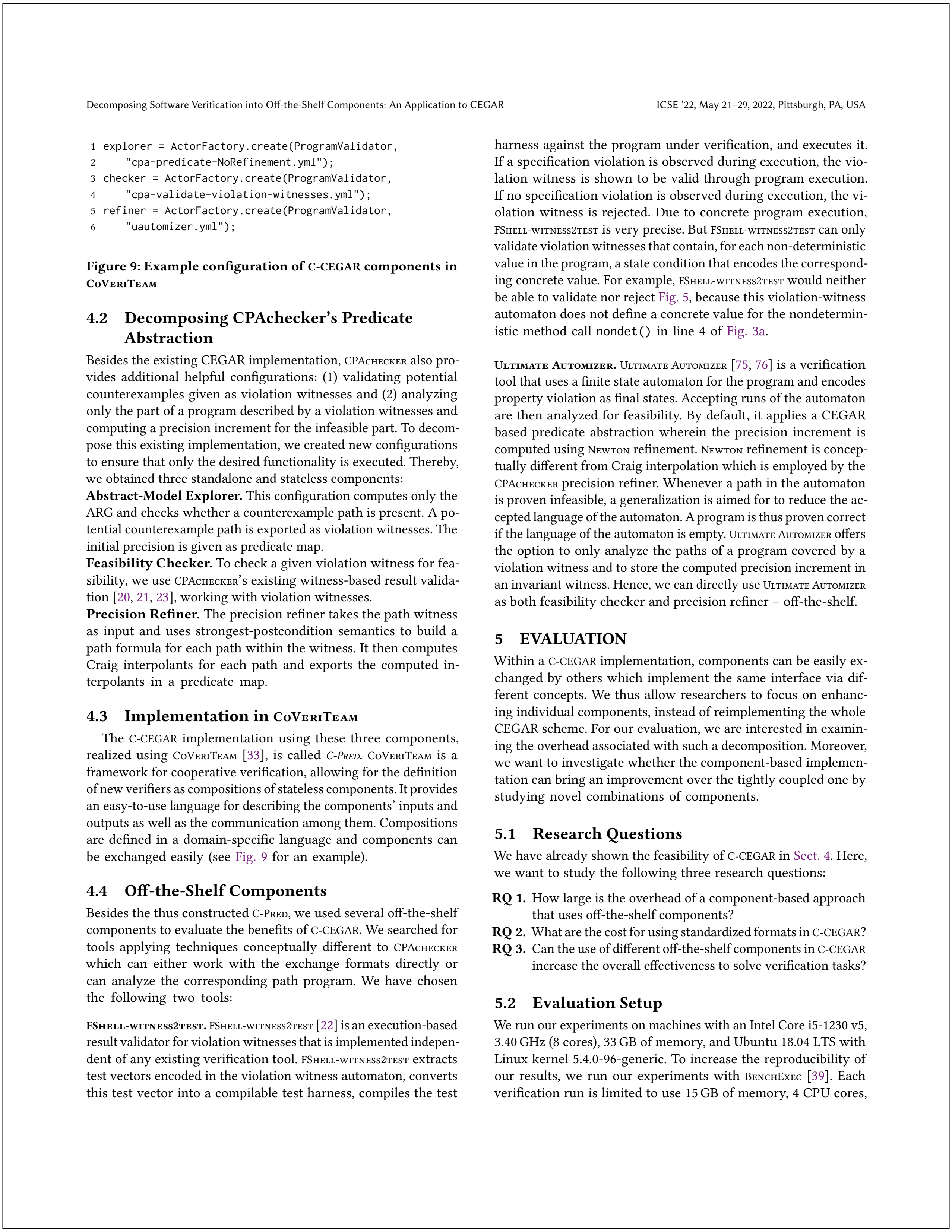
To realize a first C-CEGAR instance, we started with a decomposition of CPAchecker's implementation of CEGAR with predicate abstraction. Afterwards, we experimented with the usage of other off-the-shelf components as Feasibility Checker and Precision Refiner.

4.1 CPAchecker's Predicate Abstraction

CPAchecker [34] is a configurable tool for software verification offering many different analysis techniques, especially providing an implementation of predicate abstraction [70] using CEGAR. It has a mature code base and has proven its ability to verify and falsify programs by winning medals in the category *Overall* in SV-COMP '22 for the fifth year in a row; among others, by applying predicate abstraction.

CPAchecker's implementation of predicate abstraction (*Pred*) is a program analysis comprising two modules, a model explorer and a combined feasibility checker and precision refiner. The analysis is based on the CPA algorithm [29] with precision adjustment [30] and adjustable-block encoding (ABE) [35]. In general, the analysis information is stored in an *abstract reachability graph* (ARG), linking the analysis information with CFA nodes. The analysis stores a set of available predicates as precision for each ARG node together with a boolean formula abstracting the current state using predicates from the precision. Initially, only the predicates `true` and `false` are available. The abstraction is computed using the strongest-postcondition semantics. If the model explorer finds an (abstract) path in the ARG to an error location, this path is analyzed for feasibility.

Within Pred, a potential counterexample path is checked for feasibility by validating the path formula which is build using the strongest-postcondition semantics. It is represented in an internal format, similar and compatible with SMT-2-LIB [9]. The obtained formula is checked for satisfiability using MathSAT5 [52]. An unsatisfiable formula indicates an infeasible path. In this case, MathSAT5's Craig interpolation is used to compute a precision increment. The newly discovered predicates are added to the precision and the ARG is recomputed. Otherwise, a violation witness is computed for the found counterexample. In addition, the precision is reported using a predicate map, which is a format containing the predicates in SMT-2-LIB.



```
explorer = ActorFactory.create(ProgramValidator,
    "cpa-predicate-NoRefinement.yml");
checker = ActorFactory.create(ProgramValidator,
    "cpa-validate-violation-witnesses.yml");
refiner = ActorFactory.create(ProgramValidator,
    "uautomizer.yml");
```

Figure 9: Example configuration of C-CEGAR components in CoVeriTeam

4.2 Decomposing CPAchecker's Predicate Abstraction

Besides the existing CEGAR implementation, CPAchecker also provides additional helpful configurations: (1) validating potential counterexamples given as violation witnesses and (2) analyzing only the part of a program described by a violation witnesses and computing a precision increment for the infeasible part. To decompose this existing implementation, we created new configurations to ensure that only the desired functionality is executed. Thereby, we obtained three standalone and stateless components:

Abstract-Model Explorer. This configuration computes only the ARG and checks whether a counterexample path is present. A potential counterexample path is exported as violation witnesses. The initial precision is given as predicate map.

Feasibility Checker. To check a given violation witness for feasibility, we use CPAchecker's existing witness-based result validation [20, 21, 23], working with violation witnesses.

Precision Refiner. The precision refiner takes the path witness as input and uses strongest-postcondition semantics to build a path formula for each path within the witness. It then computes Craig interpolants for each path and exports the computed interpolants in a predicate map.

4.3 Implementation in CoVeriTeam

The C-CEGAR implementation using these three components, realized using CoVeriTeam [33], is called *C-Pred*. CoVeriTeam is a framework for cooperative verification, allowing for the definition of new verifiers as compositions of stateless components. It provides an easy-to-use language for describing the components' inputs and outputs as well as the communication among them. Compositions are defined in a domain-specific language and components can be exchanged easily (see Fig. 9 for an example).

4.4 Off-the-Shelf Components

Besides the thus constructed C-Pred, we used several off-the-shelf components to evaluate the benefits of C-CEGAR. We searched for tools applying techniques conceptually different to CPAchecker which can either work with the exchange formats directly or can analyze the corresponding path program. We have chosen the following two tools:

FShell-witness2test. FShell-witness2test [22] is an execution-based result validator for violation witnesses that is implemented independent of any existing verification tool. FShell-witness2test extracts test vectors encoded in the violation witness automaton, converts this test vector into a compilable test harness, compiles the test harness against the program under verification, and executes it. If a specification violation is observed during execution, the violation witness is shown to be valid through program execution. If no specification violation is observed during execution, the violation witness is rejected. Due to concrete program execution, FShell-witness2test is very precise. But FShell-witness2test can only validate violation witnesses that contain, for each non-deterministic value in the program, a state condition that encodes the corresponding concrete value. For example, FShell-witness2test would neither be able to validate nor reject Fig. 5, because this violation-witness automaton does not define a concrete value for the nondeterministic method call nondet() in line 4 of Fig. 3a.

Ultimate Automizer. Ultimate Automizer [75, 76] is a verification tool that uses a finite state automaton for the program and encodes property violation as final states. Accepting runs of the automaton are then analyzed for feasibility. By default, it applies a CEGAR based predicate abstraction wherein the precision increment is computed using Newton refinement. Newton refinement is conceptually different from Craig interpolation which is employed by the CPAchecker precision refiner. Whenever a path in the automaton is proven infeasible, a generalization is aimed for to reduce the accepted language of the automaton. A program is thus proven correct if the language of the automaton is empty. Ultimate Automizer offers the option to only analyze the paths of a program covered by a violation witness and to store the computed precision increment in an invariant witness. Hence, we can directly use Ultimate Automizer as both feasibility checker and precision refiner – off-the-shelf.

5 EVALUATION

Within a C-CEGAR implementation, components can be easily exchanged by others which implement the same interface via different concepts. We thus allow researchers to focus on enhancing individual components, instead of reimplementing the whole CEGAR scheme. For our evaluation, we are interested in examining the overhead associated with such a decomposition. Moreover, we want to investigate whether the component-based implementation can bring an improvement over the tightly coupled one by studying novel combinations of components.

5.1 Research Questions

We have already shown the feasibility of C-CEGAR in Sect. 4. Here, we want to study the following three research questions:

RQ 1. How large is the overhead of a component-based approach that uses off-the-shelf components?

RQ 2. What are the cost for using standardized formats in C-CEGAR?

RQ 3. Can the use of different off-the-shelf components in C-CEGAR increase the overall effectiveness to solve verification tasks?

5.2 Evaluation Setup

We run our experiments on machines with an Intel Core i5-1230 v5, 3.40 GHz (8 cores), 33 GB of memory, and Ubuntu 18.04 LTS with Linux kernel 5.4.0-96-generic. To increase the reproducibility of our results, we run our experiments with BenchExec [39]. Each verification run is limited to use 15 GB of memory, 4 CPU cores,

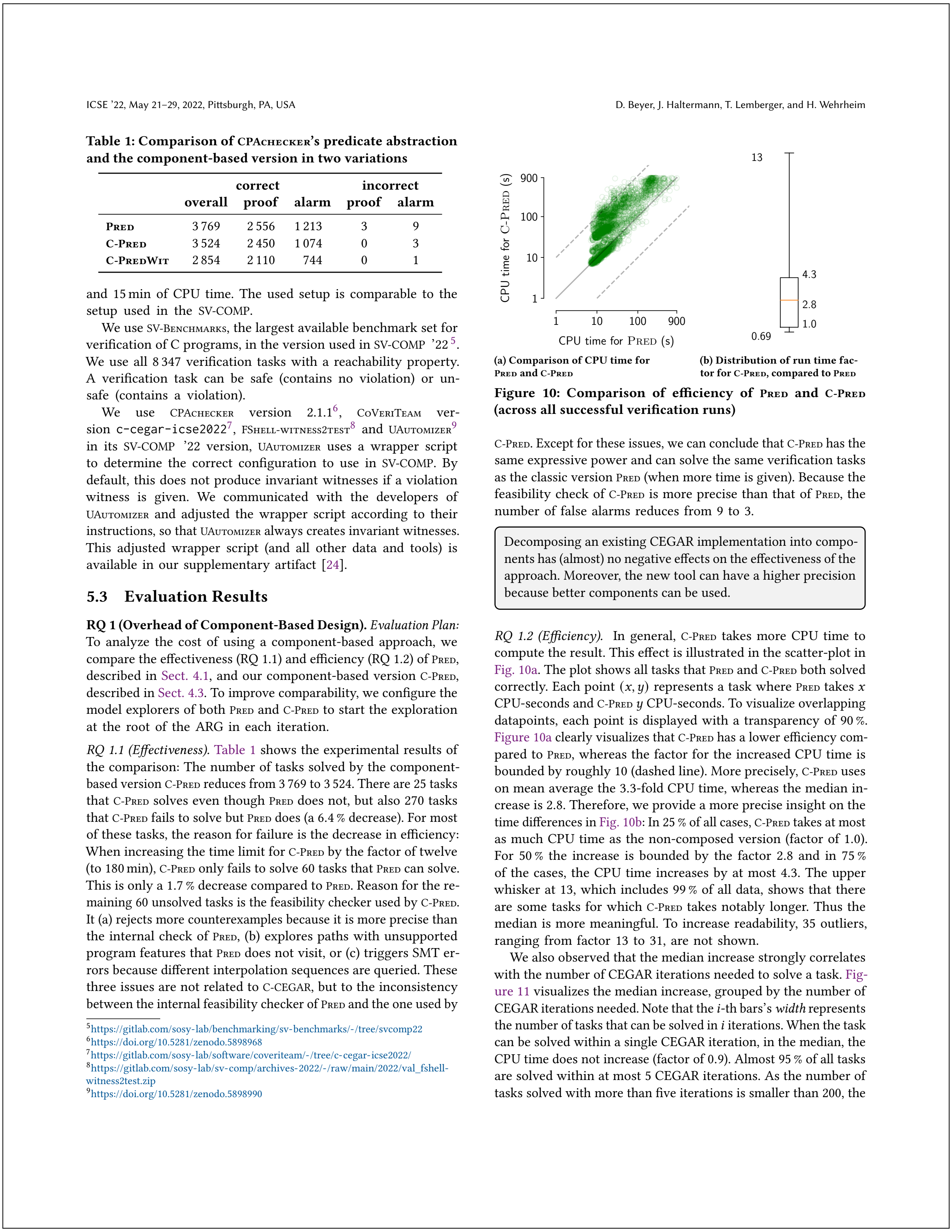


Table 1: Comparison of CPAchecker's predicate abstraction and the component-based version in two variations

		correct		incorrect	
	overall	proof	alarm	proof	alarm
Pred	3 769	2 556	1 213	3	9
C-Pred	3 524	2 450	1 074	0	3
C-PredWit	2 854	2 110	744	0	1

and 15 min of CPU time. The used setup is comparable to the setup used in the SV-COMP.

We use SV-Benchmarks, the largest available benchmark set for verification of C programs, in the version used in SV-COMP '22 [5]. We use all 8 347 verification tasks with a reachability property. A verification task can be safe (contains no violation) or unsafe (contains a violation).

We use CPAchecker version 2.1.1[6], CoVeriTeam version `c-cegar-icse2022`[7], FShell-witness2test[8] and UAutomizer[9] in its SV-COMP '22 version, UAutomizer uses a wrapper script to determine the correct configuration to use in SV-COMP. By default, this does not produce invariant witnesses if a violation witness is given. We communicated with the developers of UAutomizer and adjusted the wrapper script according to their instructions, so that UAutomizer always creates invariant witnesses. This adjusted wrapper script (and all other data and tools) is available in our supplementary artifact [24].

5.3 Evaluation Results

RQ 1 (Overhead of Component-Based Design). *Evaluation Plan:* To analyze the cost of using a component-based approach, we compare the effectiveness (RQ 1.1) and efficiency (RQ 1.2) of Pred, described in Sect. 4.1, and our component-based version C-Pred, described in Sect. 4.3. To improve comparability, we configure the model explorers of both Pred and C-Pred to start the exploration at the root of the ARG in each iteration.

RQ 1.1 (Effectiveness). Table 1 shows the experimental results of the comparison: The number of tasks solved by the component-based version C-Pred reduces from 3 769 to 3 524. There are 25 tasks that C-Pred solves even though Pred does not, but also 270 tasks that C-Pred fails to solve but Pred does (a 6.4 % decrease). For most of these tasks, the reason for failure is the decrease in efficiency: When increasing the time limit for C-Pred by the factor of twelve (to 180 min), C-Pred only fails to solve 60 tasks that Pred can solve. This is only a 1.7 % decrease compared to Pred. Reason for the remaining 60 unsolved tasks is the feasibility checker used by C-Pred. It (a) rejects more counterexamples because it is more precise than the internal check of Pred, (b) explores paths with unsupported program features that Pred does not visit, or (c) triggers SMT errors because different interpolation sequences are queried. These three issues are not related to C-CEGAR, but to the inconsistency between the internal feasibility checker of Pred and the one used by C-Pred. Except for these issues, we can conclude that C-Pred has the same expressive power and can solve the same verification tasks as the classic version Pred (when more time is given). Because the feasibility check of C-Pred is more precise than that of Pred, the number of false alarms reduces from 9 to 3.

(a) Comparison of CPU time for Pred and C-Pred

(b) Distribution of run time factor for C-Pred, compared to Pred

Figure 10: Comparison of efficiency of Pred and C-Pred (across all successful verification runs)

> Decomposing an existing CEGAR implementation into components has (almost) no negative effects on the effectiveness of the approach. Moreover, the new tool can have a higher precision because better components can be used.

RQ 1.2 (Efficiency). In general, C-Pred takes more CPU time to compute the result. This effect is illustrated in the scatter-plot in Fig. 10a. The plot shows all tasks that Pred and C-Pred both solved correctly. Each point (x, y) represents a task where Pred takes x CPU-seconds and C-Pred y CPU-seconds. To visualize overlapping datapoints, each point is displayed with a transparency of 90 %. Figure 10a clearly visualizes that C-Pred has a lower efficiency compared to Pred, whereas the factor for the increased CPU time is bounded by roughly 10 (dashed line). More precisely, C-Pred uses on mean average the 3.3-fold CPU time, whereas the median increase is 2.8. Therefore, we provide a more precise insight on the time differences in Fig. 10b: In 25 % of all cases, C-Pred takes at most as much CPU time as the non-composed version (factor of 1.0). For 50 % the increase is bounded by the factor 2.8 and in 75 % of the cases, the CPU time increases by at most 4.3. The upper whisker at 13, which includes 99 % of all data, shows that there are some tasks for which C-Pred takes notably longer. Thus the median is more meaningful. To increase readability, 35 outliers, ranging from factor 13 to 31, are not shown.

We also observed that the median increase strongly correlates with the number of CEGAR iterations needed to solve a task. Figure 11 visualizes the median increase, grouped by the number of CEGAR iterations needed. Note that the i-th bars's *width* represents the number of tasks that can be solved in i iterations. When the task can be solved within a single CEGAR iteration, in the median, the CPU time does not increase (factor of 0.9). Almost 95 % of all tasks are solved within at most 5 CEGAR iterations. As the number of tasks solved with more than five iterations is smaller than 200, the

[5] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp22

[6] https://doi.org/10.5281/zenodo.5898968

[7] https://gitlab.com/sosy-lab/software/coveriteam/-/tree/c-cegar-icse2022/

[8] https://gitlab.com/sosy-lab/sv-comp/archives-2022/-/raw/main/2022/val_fshell-witness2test.zip

[9] https://doi.org/10.5281/zenodo.5898990

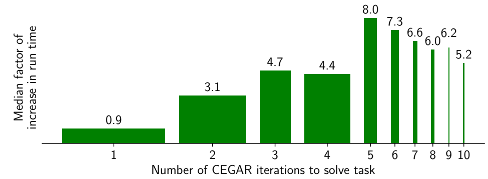
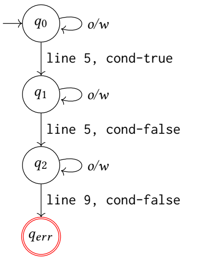


Figure 11: Median factor of run-time increase by C-Pred compared to Pred, for the first 10 numbers of CEGAR iterations. The width of the bar for i corresponds to the number of verification tasks that require exactly i CEGAR iterations

```
1  int main(void) {
2    unsigned int x = 1;
3    unsigned int y = 0;
4
5    while (y < 1024) {
6      x = 0;
7      y++;
8    }
9    if (x == 0) {}
10   else
11     error();
12 }
```



(a) Program from SV-COMP, where $x = 0$ is not a valid invariant at the loop head.

(b) Counterexample computed by model explorer

Figure 12: Example comparing predicate map and invariant witness as exchange formats

median may not perfectly summarize these iterations. We present the full figure and the raw data on our supplementary webpage [10]. The additional run time consumed by C-Pred stems mostly from the following facts: (1) Due to the three stateless components, less caching is possible (e.g. for incremental solver usage), (2) each component has to recompute basic information for the program, especially the CFA, which yields non-negligible I/O-overhead, and (3) redundant counterexample checks may be performed because feasibility check and precision refinement are fully decoupled.

The efficiency of C-Pred decreases only by a constant factor (median smaller than three).

RQ 2 (Cost of Standardized Formats).

Evaluation Plan: Instead of encoding the precision increment computed by the precision using the CPAchecker internal format predicate map, we use a standardized format, namely the invariant witness. We call this configuration C-PredWit. We compare the effectiveness and efficiency of C-Pred with C-PredWit.

Table 1 also contains the experimental results of C-PredWit. This configuration can solve in total 2 854 tasks, computing 2 110 correct proofs and 744 correct alarms. Compared to C-Pred, the effectiveness reduces by 670 tasks, a decrease of around 20 %. This decrease follows mostly from the fact that the precision refiner does not add the computed precision increment to the invariant

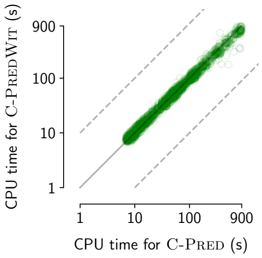


Figure 13: Comparison of run time per task of C-Pred and C-PredWit (in CPU time seconds)

witnesses. As a result, C-PredWit gets stuck in an endless loop and eventually aborts the computation.

Since invariant witnesses are not primarily designed for the exchange of a precision increment, we regularly observe this behavior. We exemplify its main reason in Fig. 12: Figure 12a contains a simplified C program from our evaluation. Before the loop body is executed for the first time, the value of x is 1 and y has the value 0. After the first loop iteration, x has the value of 0 and y's value is unequal to 0. The model explorer computes the spurious counterexample visualized in Fig. 12b. The path contains exactly one loop iteration (q_0 to q_1) and leads to the error location afterwards (q_1 to q_2 to q_{err}). A helpful precision increment which can be used to prove the counterexample to be spurious and the program to be correct contains the predicate $(y = 0 \wedge x \neq 0)$ for state q_0 and the predicate $(y \neq 0 \wedge x = 0)$ for state q_1. Although the invariant witnesses format can conceptually be used to express loop unrollings and thus can contain these two predicates, none of the precision refiners used encode these or comparable predicates in an invariant witness.

In contrast, the predicate map used to exchange information in C-Pred contains the predicates $y = 0$, $y \neq 0$, $x = 0$, and $x \neq 0$, which enable the model explorer to remove the spurious counterexample.

Next, we compare the efficiency of C-Pred and C-PredWit. Figure 13 compares the CPU time used to compute the correct solution for a task. It is visible that, except for a few outliers, both tools have the same efficiency.

The effectiveness of C-CEGAR reduces by 20 % when using standardized formats, whereas the efficiency is not influenced.

RQ 3 (Benefit of Different Components). Finally, we analyze the advantages of the component-based design by replacing the CPAchecker components by existing off-the-shelf implementations. Here, we consider two separate questions, exchanging the feasibility checker in RQ 3.1 and the precision refiner in RQ 3.2. In the following, we are using violation and invariant witnesses as exchange formats.

RQ 3.1 (Benefit of Different Feasibility Checkers). Evaluation Plan: To analyze how different feasibility checkers influence the effectiveness and efficiency, we replace CPAchecker's witness validation with both FShell-witness2test and UAutomizer. Then, we compare the effectiveness of the three resulting C-CEGAR configurations.

[10] https://www.sosy-lab.org/research/component-based-cegar/

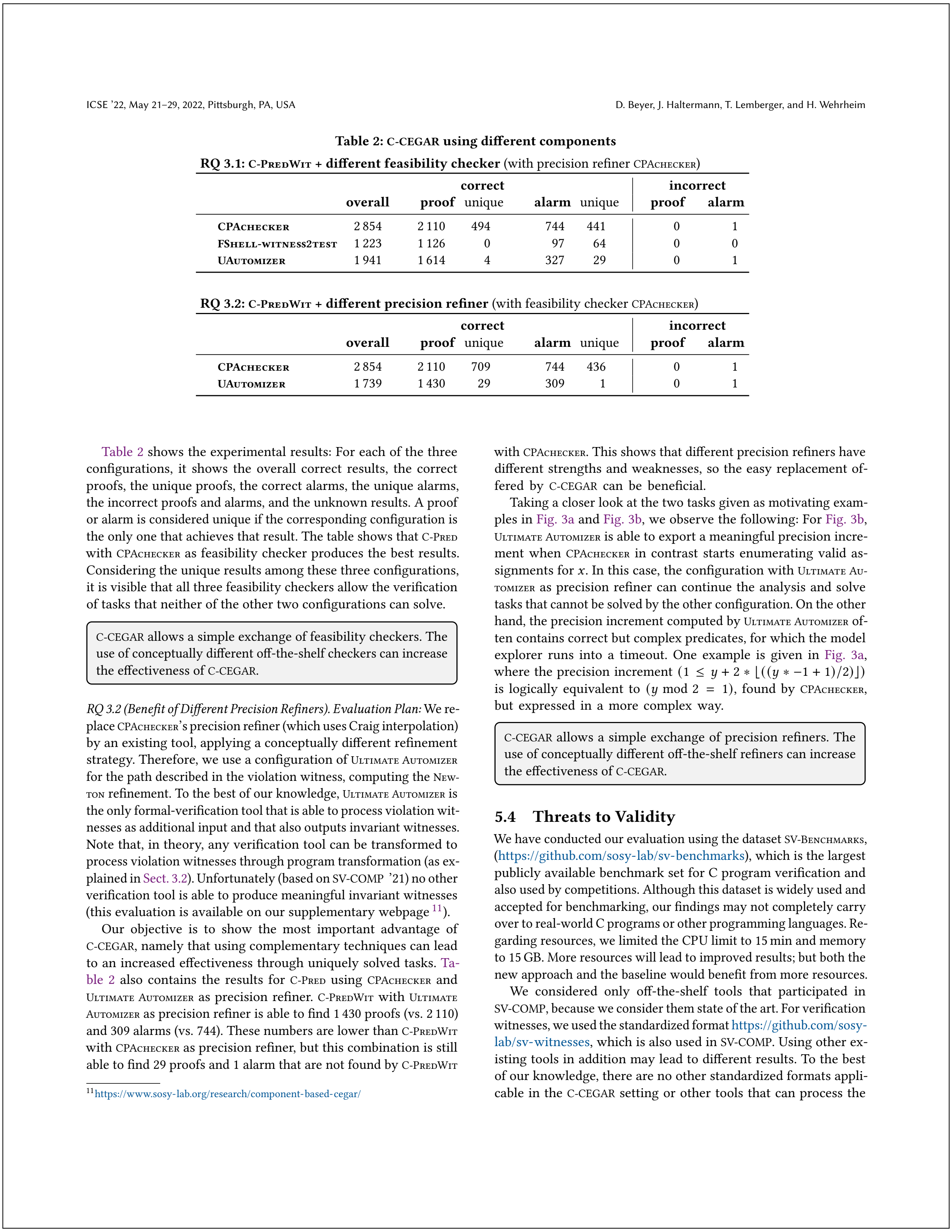


Table 2: C-CEGAR using different components

RQ 3.1: C-PredWit + different feasibility checker (with precision refiner CPAchecker)

		correct				incorrect	
	overall	**proof**	unique	**alarm**	unique	**proof**	**alarm**
CPAchecker	2 854	2 110	494	744	441	0	1
FShell-witness2test	1 223	1 126	0	97	64	0	0
UAutomizer	1 941	1 614	4	327	29	0	1

RQ 3.2: C-PredWit + different precision refiner (with feasibility checker CPAchecker)

		correct				incorrect	
	overall	**proof**	unique	**alarm**	unique	**proof**	**alarm**
CPAchecker	2 854	2 110	709	744	436	0	1
UAutomizer	1 739	1 430	29	309	1	0	1

Table 2 shows the experimental results: For each of the three configurations, it shows the overall correct results, the correct proofs, the unique proofs, the correct alarms, the unique alarms, the incorrect proofs and alarms, and the unknown results. A proof or alarm is considered unique if the corresponding configuration is the only one that achieves that result. The table shows that C-Pred with CPAchecker as feasibility checker produces the best results. Considering the unique results among these three configurations, it is visible that all three feasibility checkers allow the verification of tasks that neither of the other two configurations can solve.

C-CEGAR allows a simple exchange of feasibility checkers. The use of conceptually different off-the-shelf checkers can increase the effectiveness of C-CEGAR.

RQ 3.2 (Benefit of Different Precision Refiners). Evaluation Plan: We replace CPAchecker's precision refiner (which uses Craig interpolation) by an existing tool, applying a conceptually different refinement strategy. Therefore, we use a configuration of Ultimate Automizer for the path described in the violation witness, computing the Newton refinement. To the best of our knowledge, Ultimate Automizer is the only formal-verification tool that is able to process violation witnesses as additional input and that also outputs invariant witnesses. Note that, in theory, any verification tool can be transformed to process violation witnesses through program transformation (as explained in Sect. 3.2). Unfortunately (based on SV-COMP '21) no other verification tool is able to produce meaningful invariant witnesses (this evaluation is available on our supplementary webpage [11]).

Our objective is to show the most important advantage of C-CEGAR, namely that using complementary techniques can lead to an increased effectiveness through uniquely solved tasks. Table 2 also contains the results for C-Pred using CPAchecker and Ultimate Automizer as precision refiner. C-PredWit with Ultimate Automizer as precision refiner is able to find 1 430 proofs (vs. 2 110) and 309 alarms (vs. 744). These numbers are lower than C-PredWit with CPAchecker as precision refiner, but this combination is still able to find 29 proofs and 1 alarm that are not found by C-PredWit with CPAchecker. This shows that different precision refiners have different strengths and weaknesses, so the easy replacement offered by C-CEGAR can be beneficial.

Taking a closer look at the two tasks given as motivating examples in Fig. 3a and Fig. 3b, we observe the following: For Fig. 3b, Ultimate Automizer is able to export a meaningful precision increment when CPAchecker in contrast starts enumerating valid assignments for x. In this case, the configuration with Ultimate Automizer as precision refiner can continue the analysis and solve tasks that cannot be solved by the other configuration. On the other hand, the precision increment computed by Ultimate Automizer often contains correct but complex predicates, for which the model explorer runs into a timeout. One example is given in Fig. 3a, where the precision increment $(1 \leq y + 2 * \lfloor((y * -1 + 1)/2)\rfloor)$ is logically equivalent to $(y \bmod 2 = 1)$, found by CPAchecker, but expressed in a more complex way.

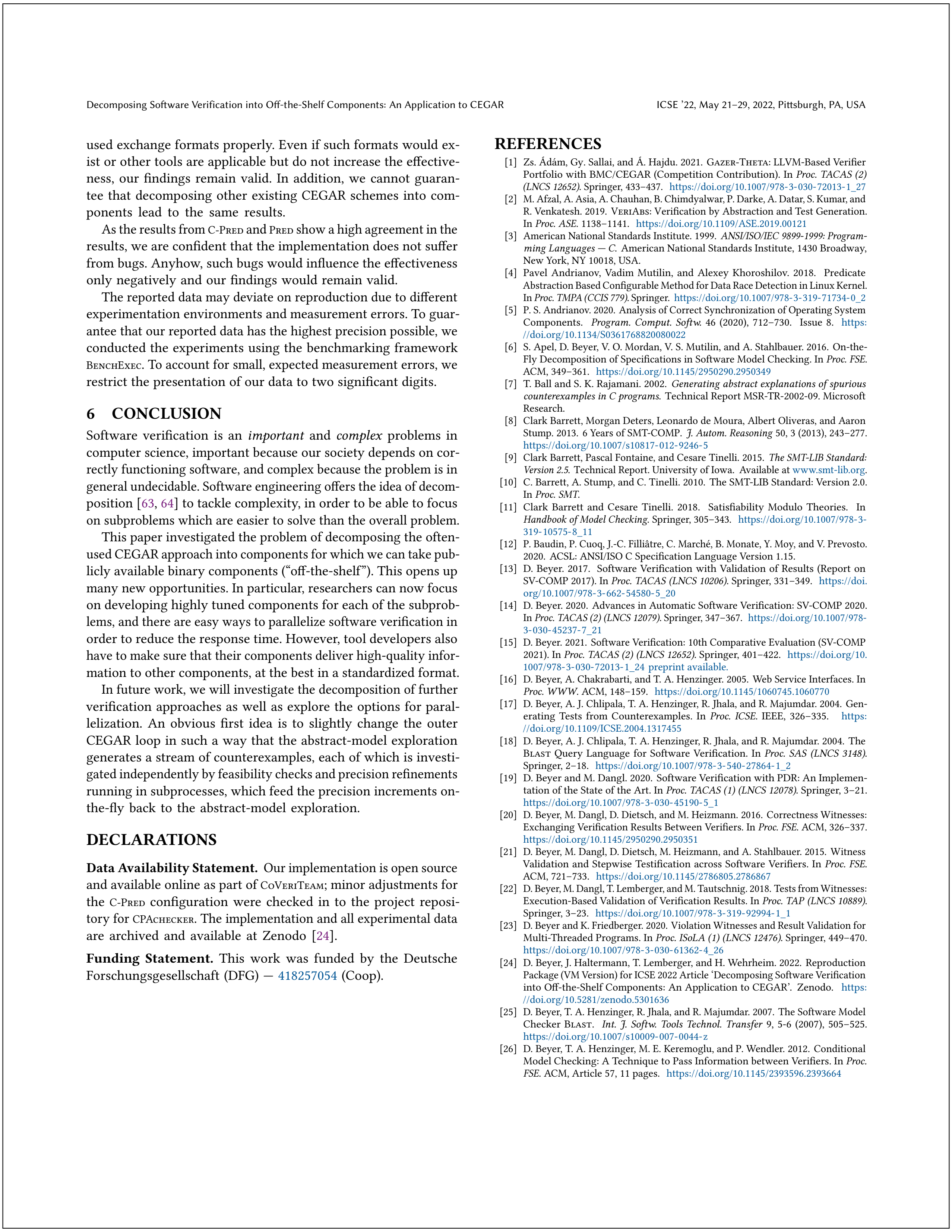
C-CEGAR allows a simple exchange of precision refiners. The use of conceptually different off-the-shelf refiners can increase the effectiveness of C-CEGAR.

5.4 Threats to Validity

We have conducted our evaluation using the dataset SV-Benchmarks, (https://github.com/sosy-lab/sv-benchmarks), which is the largest publicly available benchmark set for C program verification and also used by competitions. Although this dataset is widely used and accepted for benchmarking, our findings may not completely carry over to real-world C programs or other programming languages. Regarding resources, we limited the CPU limit to 15 min and memory to 15 GB. More resources will lead to improved results; but both the new approach and the baseline would benefit from more resources.

We considered only off-the-shelf tools that participated in SV-COMP, because we consider them state of the art. For verification witnesses, we used the standardized format https://github.com/sosy-lab/sv-witnesses, which is also used in SV-COMP. Using other existing tools in addition may lead to different results. To the best of our knowledge, there are no other standardized formats applicable in the C-CEGAR setting or other tools that can process the

[11] https://www.sosy-lab.org/research/component-based-cegar/



used exchange formats properly. Even if such formats would exist or other tools are applicable but do not increase the effectiveness, our findings remain valid. In addition, we cannot guarantee that decomposing other existing CEGAR schemes into components lead to the same results.

As the results from C-Pred and Pred show a high agreement in the results, we are confident that the implementation does not suffer from bugs. Anyhow, such bugs would influence the effectiveness only negatively and our findings would remain valid.

The reported data may deviate on reproduction due to different experimentation environments and measurement errors. To guarantee that our reported data has the highest precision possible, we conducted the experiments using the benchmarking framework BenchExec. To account for small, expected measurement errors, we restrict the presentation of our data to two significant digits.

6 CONCLUSION

Software verification is an *important* and *complex* problems in computer science, important because our society depends on correctly functioning software, and complex because the problem is in general undecidable. Software engineering offers the idea of decomposition [63, 64] to tackle complexity, in order to be able to focus on subproblems which are easier to solve than the overall problem.

This paper investigated the problem of decomposing the often-used CEGAR approach into components for which we can take publicly available binary components ("off-the-shelf"). This opens up many new opportunities. In particular, researchers can now focus on developing highly tuned components for each of the subproblems, and there are easy ways to parallelize software verification in order to reduce the response time. However, tool developers also have to make sure that their components deliver high-quality information to other components, at the best in a standardized format.

In future work, we will investigate the decomposition of further verification approaches as well as explore the options for parallelization. An obvious first idea is to slightly change the outer CEGAR loop in such a way that the abstract-model exploration generates a stream of counterexamples, each of which is investigated independently by feasibility checks and precision refinements running in subprocesses, which feed the precision increments on-the-fly back to the abstract-model exploration.

DECLARATIONS

Data Availability Statement. Our implementation is open source and available online as part of CoVeriTeam; minor adjustments for the C-Pred configuration were checked in to the project repository for CPAchecker. The implementation and all experimental data are archived and available at Zenodo [24].

Funding Statement. This work was funded by the Deutsche Forschungsgesellschaft (DFG) — 418257054 (Coop).

REFERENCES

[1] Zs. Ádám, Gy. Sallai, and Á. Hajdu. 2021. Gazer-Theta: LLVM-Based Verifier Portfolio with BMC/CEGAR (Competition Contribution). In *Proc. TACAS (2) (LNCS 12652)*. Springer, 433–437. https://doi.org/10.1007/978-3-030-72013-1_27

[2] M. Afzal, A. Asia, A. Chauhan, B. Chimdyalwar, P. Darke, A. Datar, S. Kumar, and R. Venkatesh. 2019. VeriAbs: Verification by Abstraction and Test Generation. In *Proc. ASE*. 1138–1141. https://doi.org/10.1109/ASE.2019.00121

[3] American National Standards Institute. 1999. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA.

[4] Pavel Andrianov, Vadim Mutilin, and Alexey Khoroshilov. 2018. Predicate Abstraction Based Configurable Method for Data Race Detection in Linux Kernel. In *Proc. TMPA (CCIS 779)*. Springer. https://doi.org/10.1007/978-3-319-71734-0_2

[5] P. S. Andrianov. 2020. Analysis of Correct Synchronization of Operating System Components. *Program. Comput. Softw.* 46 (2020), 712–730. Issue 8. https://doi.org/10.1134/S0361768820080022

[6] S. Apel, D. Beyer, V. O. Mordan, V. S. Mutilin, and A. Stahlbauer. 2016. On-the-Fly Decomposition of Specifications in Software Model Checking. In *Proc. FSE*. ACM, 349–361. https://doi.org/10.1145/2950290.2950349

[7] T. Ball and S. K. Rajamani. 2002. *Generating abstract explanations of spurious counterexamples in C programs*. Technical Report MSR-TR-2002-09. Microsoft Research.

[8] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 2013. 6 Years of SMT-COMP. *J. Autom. Reasoning* 50, 3 (2013), 243–277. https://doi.org/10.1007/s10817-012-9246-5

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. *The SMT-LIB Standard: Version 2.5*. Technical Report. University of Iowa. Available at www.smt-lib.org.

[10] C. Barrett, A. Stump, and C. Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proc. SMT*.

[11] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Springer, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[12] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. 2020. ACSL: ANSI/ISO C Specification Language Version 1.15.

[13] D. Beyer. 2017. Software Verification with Validation of Results (Report on SV-COMP 2017). In *Proc. TACAS (LNCS 10206)*. Springer, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[14] D. Beyer. 2020. Advances in Automatic Software Verification: SV-COMP 2020. In *Proc. TACAS (2) (LNCS 12079)*. Springer, 347–367. https://doi.org/10.1007/978-3-030-45237-7_21

[15] D. Beyer. 2021. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Proc. TACAS (2) (LNCS 12652)*. Springer, 401–422. https://doi.org/10.1007/978-3-030-72013-1_24 preprint available.

[16] D. Beyer, A. Chakrabarti, and T. A. Henzinger. 2005. Web Service Interfaces. In *Proc. WWW*. ACM, 148–159. https://doi.org/10.1145/1060745.1060770

[17] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating Tests from Counterexamples. In *Proc. ICSE*. IEEE, 326–335. https://doi.org/10.1109/ICSE.2004.1317455

[18] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. The Blast Query Language for Software Verification. In *Proc. SAS (LNCS 3148)*. Springer, 2–18. https://doi.org/10.1007/978-3-540-27864-1_2

[19] D. Beyer and M. Dangl. 2020. Software Verification with PDR: An Implementation of the State of the Art. In *Proc. TACAS (1) (LNCS 12078)*. Springer, 3–21. https://doi.org/10.1007/978-3-030-45190-5_1

[20] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

[21] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. FSE*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867

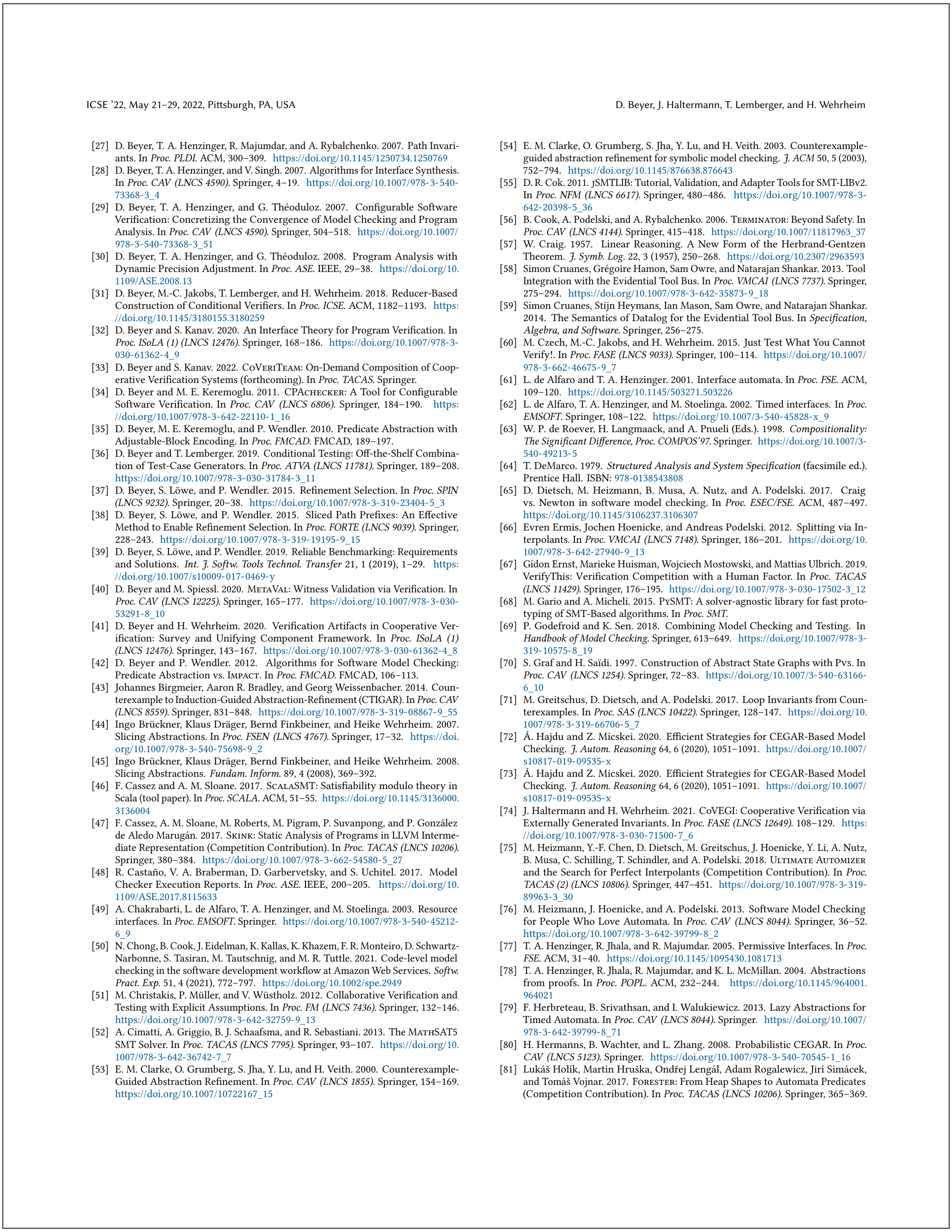
[22] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. 2018. Tests from Witnesses: Execution-Based Validation of Verification Results. In *Proc. TAP (LNCS 10889)*. Springer, 3–23. https://doi.org/10.1007/978-3-319-92994-1_1

[23] D. Beyer and K. Friedberger. 2020. Violation Witnesses and Result Validation for Multi-Threaded Programs. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 449–470. https://doi.org/10.1007/978-3-030-61362-4_26

[24] D. Beyer, J. Haltermann, T. Lemberger, and H. Wehrheim. 2022. Reproduction Package (VM Version) for ICSE 2022 Article 'Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR'. Zenodo. https://doi.org/10.5281/zenodo.5301636

[25] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The Software Model Checker Blast. *Int. J. Softw. Tools Technol. Transfer* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[26] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *Proc. FSE*. ACM, Article 57, 11 pages. https://doi.org/10.1145/2393596.2393664



[27] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. 2007. Path Invariants. In *Proc. PLDI*. ACM, 300–309. https://doi.org/10.1145/1250734.1250769

[28] D. Beyer, T. A. Henzinger, and V. Singh. 2007. Algorithms for Interface Synthesis. In *Proc. CAV (LNCS 4590)*. Springer, 4–19. https://doi.org/10.1007/978-3-540-73368-3_4

[29] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[30] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2008. Program Analysis with Dynamic Precision Adjustment. In *Proc. ASE*. IEEE, 29–38. https://doi.org/10.1109/ASE.2008.13

[31] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. 2018. Reducer-Based Construction of Conditional Verifiers. In *Proc. ICSE*. ACM, 1182–1193. https://doi.org/10.1145/3180155.3180259

[32] D. Beyer and S. Kanav. 2020. An Interface Theory for Program Verification. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 168–186. https://doi.org/10.1007/978-3-030-61362-4_9

[33] D. Beyer and S. Kanav. 2022. CoVeriTeam: On-Demand Composition of Cooperative Verification Systems (forthcoming). In *Proc. TACAS*. Springer.

[34] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proc. CAV (LNCS 6806)*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[35] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197.

[36] D. Beyer and T. Lemberger. 2019. Conditional Testing: Off-the-Shelf Combination of Test-Case Generators. In *Proc. ATVA (LNCS 11781)*. Springer, 189–208. https://doi.org/10.1007/978-3-030-31784-3_11

[37] D. Beyer, S. Löwe, and P. Wendler. 2015. Refinement Selection. In *Proc. SPIN (LNCS 9232)*. Springer, 20–38. https://doi.org/10.1007/978-3-319-23404-5_3

[38] D. Beyer, S. Löwe, and P. Wendler. 2015. Sliced Path Prefixes: An Effective Method to Enable Refinement Selection. In *Proc. FORTE (LNCS 9039)*. Springer, 228–243. https://doi.org/10.1007/978-3-319-19195-9_15

[39] D. Beyer, S. Löwe, and P. Wendler. 2019. Reliable Benchmarking: Requirements and Solutions. *Int. J. Softw. Tools Technol. Transfer* 21, 1 (2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y

[40] D. Beyer and M. Spiessl. 2020. MetaVal: Witness Validation via Verification. In *Proc. CAV (LNCS 12225)*. Springer, 165–177. https://doi.org/10.1007/978-3-030-53291-8_10

[41] D. Beyer and H. Wehrheim. 2020. Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 143–167. https://doi.org/10.1007/978-3-030-61362-4_8

[42] D. Beyer and P. Wendler. 2012. Algorithms for Software Model Checking: Predicate Abstraction vs. Impact. In *Proc. FMCAD*. FMCAD, 106–113.

[43] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. 2014. Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR). In *Proc. CAV (LNCS 8559)*. Springer, 831–848. https://doi.org/10.1007/978-3-319-08867-9_55

[44] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. 2007. Slicing Abstractions. In *Proc. FSEN (LNCS 4767)*. Springer, 17–32. https://doi.org/10.1007/978-3-540-75698-9_2

[45] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. 2008. Slicing Abstractions. *Fundam. Inform.* 89, 4 (2008), 369–392.

[46] F. Cassez and A. M. Sloane. 2017. ScalaSMT: Satisfiability modulo theory in Scala (tool paper). In *Proc. SCALA*. ACM, 51–55. https://doi.org/10.1145/3136000.3136004

[47] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. González de Aledo Marugán. 2017. Skink: Static Analysis of Programs in LLVM Intermediate Representation (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 380–384. https://doi.org/10.1007/978-3-662-54580-5_27

[48] R. Castaño, V. A. Braberman, D. Garbervetsky, and S. Uchitel. 2017. Model Checker Execution Reports. In *Proc. ASE*. IEEE, 200–205. https://doi.org/10.1109/ASE.2017.8115633

[49] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. 2003. Resource interfaces. In *Proc. EMSOFT*. Springer. https://doi.org/10.1007/978-3-540-45212-6_9

[50] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle. 2021. Code-level model checking in the software development workflow at Amazon Web Services. *Softw. Pract. Exp.* 51, 4 (2021), 772–797. https://doi.org/10.1002/spe.2949

[51] M. Christakis, P. Müller, and V. Wüstholz. 2012. Collaborative Verification and Testing with Explicit Assumptions. In *Proc. FM (LNCS 7436)*. Springer, 132–146. https://doi.org/10.1007/978-3-642-32759-9_13

[52] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proc. TACAS (LNCS 7795)*. Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

[53] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Proc. CAV (LNCS 1855)*. Springer, 154–169. https://doi.org/10.1007/10722167_15

[54] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[55] D. R. Cok. 2011. jSMTLIB: Tutorial, Validation, and Adapter Tools for SMT-LIBv2. In *Proc. NFM (LNCS 6617)*. Springer, 480–486. https://doi.org/10.1007/978-3-642-20398-5_36

[56] B. Cook, A. Podelski, and A. Rybalchenko. 2006. Terminator: Beyond Safety. In *Proc. CAV (LNCS 4144)*. Springer, 415–418. https://doi.org/10.1007/11817963_37

[57] W. Craig. 1957. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *J. Symb. Log.* 22, 3 (1957), 250–268. https://doi.org/10.2307/2963593

[58] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. 2013. Tool Integration with the Evidential Tool Bus. In *Proc. VMCAI (LNCS 7737)*. Springer, 275–294. https://doi.org/10.1007/978-3-642-35873-9_18

[59] Simon Cruanes, Stijn Heymans, Ian Mason, Sam Owre, and Natarajan Shankar. 2014. The Semantics of Datalog for the Evidential Tool Bus. In *Specification, Algebra, and Software*. Springer, 256–275.

[60] M. Czech, M.-C. Jakobs, and H. Wehrheim. 2015. Just Test What You Cannot Verify!. In *Proc. FASE (LNCS 9033)*. Springer, 100–114. https://doi.org/10.1007/978-3-662-46675-9_7

[61] L. de Alfaro and T. A. Henzinger. 2001. Interface automata. In *Proc. FSE*. ACM, 109–120. https://doi.org/10.1145/503271.503226

[62] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. 2002. Timed interfaces. In *Proc. EMSOFT*. Springer, 108–122. https://doi.org/10.1007/3-540-45828-x_9

[63] W. P. de Roever, H. Langmaack, and A. Pnueli (Eds.). 1998. *Compositionality: The Significant Difference, Proc. COMPOS'97*. Springer. https://doi.org/10.1007/3-540-49213-5

[64] T. DeMarco. 1979. *Structured Analysis and System Specification* (facsimile ed.). Prentice Hall. ISBN: 978-0138543808

[65] D. Dietsch, M. Heizmann, B. Musa, A. Nutz, and A. Podelski. 2017. Craig vs. Newton in software model checking. In *Proc. ESEC/FSE*. ACM, 487–497. https://doi.org/10.1145/3106237.3106307

[66] Evren Ermis, Jochen Hoenicke, and Andreas Podelski. 2012. Splitting via Interpolants. In *Proc. VMCAI (LNCS 7148)*. Springer, 186–201. https://doi.org/10.1007/978-3-642-27940-9_13

[67] Gidon Ernst, Marieke Huisman, Wojciech Mostowski, and Mattias Ulbrich. 2019. VerifyThis: Verification Competition with a Human Factor. In *Proc. TACAS (LNCS 11429)*. Springer, 176–195. https://doi.org/10.1007/978-3-030-17502-3_12

[68] M. Gario and A. Micheli. 2015. PySMT: A solver-agnostic library for fast prototyping of SMT-Based algorithms. In *Proc. SMT*.

[69] P. Godefroid and K. Sen. 2018. Combining Model Checking and Testing. In *Handbook of Model Checking*. Springer, 613–649. https://doi.org/10.1007/978-3-319-10575-8_19

[70] S. Graf and H. Saïdi. 1997. Construction of Abstract State Graphs with Pvs. In *Proc. CAV (LNCS 1254)*. Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10

[71] M. Greitschus, D. Dietsch, and A. Podelski. 2017. Loop Invariants from Counterexamples. In *Proc. SAS (LNCS 10422)*. Springer, 128–147. https://doi.org/10.1007/978-3-319-66706-5_7

[72] Á. Hajdu and Z. Micskei. 2020. Efficient Strategies for CEGAR-Based Model Checking. *J. Autom. Reasoning* 64, 6 (2020), 1051–1091. https://doi.org/10.1007/s10817-019-09535-x

[73] Á. Hajdu and Z. Micskei. 2020. Efficient Strategies for CEGAR-Based Model Checking. *J. Autom. Reasoning* 64, 6 (2020), 1051–1091. https://doi.org/10.1007/s10817-019-09535-x

[74] J. Haltermann and H. Wehrheim. 2021. CoVEGI: Cooperative Verification via Externally Generated Invariants. In *Proc. FASE (LNCS 12649)*. 108–129. https://doi.org/10.1007/978-3-030-71500-7_6

[75] M. Heizmann, Y.-F. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. 2018. Ultimate Automizer and the Search for Perfect Interpolants (Competition Contribution). In *Proc. TACAS (2) (LNCS 10806)*. Springer, 447–451. https://doi.org/10.1007/978-3-319-89963-3_30

[76] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[77] T. A. Henzinger, R. Jhala, and R. Majumdar. 2005. Permissive Interfaces. In *Proc. FSE*. ACM, 31–40. https://doi.org/10.1145/1095430.1081713

[78] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. 2004. Abstractions from proofs. In *Proc. POPL*. ACM, 232–244. https://doi.org/10.1145/964001.964021

[79] F. Herbreteau, B. Srivathsan, and I. Walukiewicz. 2013. Lazy Abstractions for Timed Automata. In *Proc. CAV (LNCS 8044)*. Springer. https://doi.org/10.1007/978-3-642-39799-8_71

[80] H. Hermanns, B. Wachter, and L. Zhang. 2008. Probabilistic CEGAR. In *Proc. CAV (LNCS 5123)*. Springer. https://doi.org/10.1007/978-3-540-70545-1_16

[81] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jirí Simácek, and Tomáš Vojnar. 2017. Forester: From Heap Shapes to Automata Predicates (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 365–369.

https://doi.org/10.1007/978-3-662-54580-5_24

[82] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (2012). https://doi.org/10.1609/aimag.v33i1.2395

[83] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Proc. CAV (LNCS 9779)*. Springer, 352–358. https://doi.org/10.1007/978-3-319-41528-4_19

[84] E. G. Karpenkov, K. Friedberger, and D. Beyer. 2016. JavaSMT: A Unified Interface for SMT Solvers in Java. In *Proc. VSTTE (LNCS 9971)*. Springer, 139–148. https://doi.org/10.1007/978-3-319-48869-1_11

[85] M. Mann, A. Wilson, C. Tinelli, and C. W. Barrett. 2020. SMT-Switch: A solver-agnostic C++ API for SMT Solving. *arXiv/CoRR* 2007.01374 (2020). arXiv:2007.01374 https://arxiv.org/abs/2007.01374.

[86] Tiziana Margaria. 2005. Web services-Based tool-integration in the ETI platform. *Software and Systems Modeling* 4, 2 (2005), 141–156. https://doi.org/10.1007/s10270-004-0072-z

[87] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. 2005. jETI: A Tool for Remote Tool Integration. In *Proc. TACAS (LNCS 3440)*. Springer, 557–562. https://doi.org/10.1007/978-3-540-31980-1_38

[88] T. Margaria, R. Nagel, and B. Steffen. 2005. Remote integration and coordination of verification tools in jETI. In *Proc. ECBS*. 431–436. https://doi.org/10.1109/ECBS.2005.59

[89] K. L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Proc. CAV (LNCS 2725)*. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[90] K. L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proc. CAV (LNCS 4144)*. Springer, 123–136. https://doi.org/10.1007/11817963_14

[91] F. Pauck and H. Wehrheim. 2019. Together Strong: Cooperative Android App Analysis. In *Proc. ESEC/FSE*. ACM, 374–384. https://doi.org/10.1145/3338906.3338915

[92] Nir Piterman and Amir Pnueli. 2018. Temporal Logic and Fair Discrete Systems. In *Handbook of Model Checking*. Springer, 27–73. https://doi.org/10.1007/978-3-319-10575-8_2

[93] A. Podelski and A. Rybalchenko. 2005. Transition predicate abstraction and fair termination. In *Proc. POPL*. ACM, 132–144. https://doi.org/10.1145/1040305.1040317

[94] Z. Rakamarić and M. Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proc. CAV (LNCS 8559)*. Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[95] C. Richter, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim. 2020. Algorithm selection for software validation based on graph kernels. *Autom. Softw. Eng.* 27, 1 (2020), 153–186. https://doi.org/10.1007/s10515-020-00270-x

[96] C. Richter and H. Wehrheim. 2019. PeSCo: Predicting Sequential Combinations of Verifiers (Competition Contribution). In *Proc. TACAS (3) (LNCS 11429)*. Springer, 229–233. https://doi.org/10.1007/978-3-030-17502-3_19

[97] John M. Rushby. 2005. An Evidential Tool Bus. In *Proc. ICFEM (LNCS 3785)*. Springer, 36–36. https://doi.org/10.1007/11576280_3

[98] N. Shankar. 2018. Combining Model Checking and Deduction. In *Handbook of Model Checking*. Springer, 651–684. https://doi.org/10.1007/978-3-319-10575-8_20

[99] Bernhard Steffen, Tiziana Margaria, and Volker Braun. 1997. The Electronic Tool Integration Platform: Concepts and Design. *STTT* 1, 1-2 (1997), 9–30. https://doi.org/10.1007/s100090050003

[100] Cong Tian, Zhenhua Duan, and Zhao Duan. 2014. Making CEGAR More Efficient in Software Model Checking. *IEEE Trans. Softw. Eng.* 40, 12 (2014), 1206–1223. https://doi.org/10.1109/TSE.2014.2357442

[101] Anton R. Volkov and Mikhail U. Mandrykin. 2017. Predicate Abstractions Memory Modeling Method with Separation into Disjoint Regions. *Proceedings of the Institute for System Programming (ISPRAS)* 29 (2017), 203–216. Issue 4. https://doi.org/10.15514/ISPRAS-2017-29(4)-13

[102] D. Wang, C. Zhang, G. Chen, M. Gu, and J. Sun. 2016. C Code Verification based on the Extended Labeled Transition System Model. In *MoDELS 2016 (CEUR 1725)*. CEUR-WS.org, 48–55.

[103] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2018. On Scheduling Constraint Abstraction for Multi-Threaded Program Verification. *IEEE Trans. Softw. Eng.* (2018). https://doi.org/10.1109/TSE.2018.2864122