Witness-Based Validation of Verification Results with Applications to Software-Model Checking

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

eingereicht von

Matthias Dangl

9. März 2022

1. Gutachter:	Prof. Dr. Dirk Beyer
2. Gutachter:	Prof. Dr. Stefan Leue
3. Gutachter:	Prof. Dr. Andreas Podelski
Tag der mündlichen Prüfung:	15.12.2022

Abstract

In the scientific world, formal verification is an established engineering technique to ensure the correctness of hardware and software systems. Because formal verification is an arduous and error-prone endeavor, automated solutions are desirable, and researchers continue to develop new algorithms and optimize existing ones to push the boundaries of what can be verified automatically. These efforts do not go unnoticed by the industry. Hardware-circuit designs, flight-control systems, and operating-system drivers are just a few examples of systems where formal verification is already part of the quality-assurance repertoire. Nevertheless, the primary fields of application for formal verification are mainly those where errors carry a high risk of significant damage, either financial or physical, because the costs of formal verification are considered to be too high for most other projects, despite the fact that the research community has made vast advancements regarding the effectiveness and efficiency of formal verification techniques in the last decades. We present and address two potential reasons for this discrepancy that we identified in the field of automated formal software verification.

(1) Even for experts in the field, it is often difficult to decide which of the multitude of available techniques is the most suitable solution they should recommend to solve a given verification problem. Moreover, even if a suitable solution is found for a given system, there is no guarantee that the solution is sustainable as the system evolves. Consequently, the cost of finding and maintaining a suitable approach for applying formal software verification to real-world systems is high. (2) Even assuming that a suitable and maintainable solution for applying formal software verification to a given system is found and verification results could be obtained, developers of the system still require further guidance towards making practical use of these results, which often differ significantly from the results they obtain from classical quality-assurance techniques they are familiar with, such as testing.

To mitigate the first issue, using the open-source software-verification framework CPAchecker, we investigate several popular formal software-verification techniques such as predicate abstraction, Impact, bounded model checking, k-induction, and PDR, and perform an extensive and rigorous experimental study to identify their strengths and weaknesses regarding their comparative effectiveness and efficiency when applied to a large and established benchmark set, to provide a basis for choosing the best technique for a given problem.

To mitigate the second issue, we propose a concrete standard format for the representation and communication of verification results that raises the bar from plain "yes" or "no" answers to verification witnesses, which are valuable artifacts of the verification process that contain detailed information discovered during the analysis. We then use these verification witnesses for several applications: To increase the trust in verification results, we first develop several independent validators based on violation witnesses, i.e. verification witnesses that represent bugs detected by a verifier. We then extend our validators to also verify the verification results obtained from a successful verification, which are represented by correctness witnesses. Lastly, we also develop an interactive web service to store and retrieve these verification witnesses, to provide online validation to quickly de-prioritize likely wrong results, and to graphically visualize the witnesses, as an example of how verification can be integrated into a development process. Since the introduction of our

proposed standard format for verification witnesses, it has been adopted by over thirty different software verifiers, and our witness-based result-validation tools have become a core component in the scoring process of the International Competition on Software Verification.

Zusammenfassung

In der Welt der Wissenschaft gilt die Formale Verifikation als etablierte Methode, die Korrektheit von Hard- und Software zu gewährleisten. Da die Anwendung formaler Verifikation jedoch selbst ein beschwerliches und fehlerträchtiges Unterfangen darstellt, ist es erstrebenswert, automatisierte Lösungen dafür zu finden. Forscher entwickeln daher immer wieder neue Algorithmen Formaler Verifikation oder verbessern bereits existierende Algorithmen, um die Grenzen der Automatisierbarkeit Formaler Verifikation weiter und weiter zu dehnen. Auch die Industrie ist bereits auf diese Anstrengungen aufmerksam geworden. Flugsteuerungssysteme, Betriebssystemtreiber und Entwürfe von Hardware-Schaltungen sind nur einzelne Beispiele von Systemen, bei denen Formale Verifikation bereits heute einen festen Stammplatz im Arsenal der Qualitätssicherungsmaßnahmen eingenommen hat. Trotz alledem bleiben die primären Einsatzgebiete Formaler Verifikation jene, in denen Fehler ein hohes Risiko finanzieller oder physischer Schäden bergen, da in anderen Projekten die Kosten des Einsatzes Formaler Verifikation in der Regel als zu hoch empfunden werden, unbeachtet der Tatsache, dass es der Forschungsgemeinschaft in den letzten Jahrzehnten gelungen ist, enorme Fortschritte bei der Verbesserung der Effektivität und Effizienz Formaler Verifikationstechniken zu machen. Wir präsentieren und diskutieren zwei potenzielle Ursachen für diese Diskrepanz zwischen Forschung und Industrie, die wir auf dem Gebiet der Automatisierten Formalen Softwareverifikation identifiziert haben.

(1) Sogar Fachleuten fällt es oft schwer, zu entscheiden, welche der zahlreichen verfügbaren Methoden sie als vielversprechendste Lösung eines gegebenen Verifikationsproblems empfehlen sollten. Darüber hinaus gibt es selbst dann, wenn eine passende Lösung für ein gegebenes System gefunden wird, keine Garantie, dass sich diese Lösung im Laufe der Evolution des Systems als Nachhaltig erweisen wird. Daher sind sowohl die Wahl als auch der Unterhalt eines passenden Ansatzes zur Anwendung Formaler Softwareverifikation auf reale Systeme kostspielige Unterfangen. (2) Selbst unter der Annahme, dass eine passende und wartbare Lösung zur Anwendung Formaler Softwareverifikation auf ein gegebenes System gefunden und Verifikationsergebnisse erzielt werden, benötigen die Entwickler des Systems immer noch weitere Unterstützung, um einen praktischen Nutzen aus den Ergebnissen ziehen zu können, die sich oft maßgeblich unterscheiden von den Ergebnissen jener klassischen Qualitätssicherungssysteme, mit denen sie vertraut sind, wie beispielsweise dem Testen.

Um das erste Problem zu entschärfen, untersuchen wir unter Verwendung des Open-Source-Softwareverifikationsystems CPAchecker mehrere beliebte Formale Softwareverifikationsmethoden, wie beispielsweise Prädikatenabstraktion, Impact, Bounded-Model-Checking, k-Induktion und PDR, und führen umfangreiche und gründliche experimentelle Studien auf einem großen und etablierten Konvolut an Beispielprogrammen durch, um die Stärken und Schwächen dieser Methoden hinsichtlich ihrer relativen Effektivität und Effizienz zu ermitteln und daraus eine Entscheidungsgrundlage für die Wahl der besten Lösung für ein gegebenes Problem abzuleiten.

Um das zweite Problem zu entschärfen, schlagen wir ein konkretes Standardformat zur Modellierung und zum Austausch von Verifikationsergebnissen vor, welches die Ansprüche an Verifikationsergebnisse anhebt, weg von einfachen "ja/nein"-Antworten und hin zu Verifikationszeugen (Verification Witnesses), bei denen es sich um wertvolle Produkte

des Verifikationsprozesses handelt und die detaillierte, während der Analyse entdeckte Informationen enthalten. Wir stellen mehrere Anwendungsbeispiele für diese Verifikationszeugen vor: Um das Vertrauen in Verifikationsergebnisse zu erhöhen, entwickeln wir zunächst mehrere, voneinander unabhängige Validatoren, die Verletzungszeugen (Violation Witnesses) verwenden, also Verifikationszeugen, welche von einem Verifikationswerkzeug gefundene Spezifikationsverletzungen darstellen, Diese Validatoren erweitern wir anschließend so, dass sie auch in der Lage sind, die Verifikationsergebnisse erfolgreicher Verifikationen, also Korrektheitsbehauptungen, die durch Korrektheitszeugen (Correctness Witnesses) dokumentiert werden, nachzuvollziehen. Schlussendlich entwickeln wir als Beispiel für die Integrierbarkeit Formaler Verifikation in den Entwicklungsprozess einen interaktiven Webservice für die Speicherung und den Abruf von Verifikationzeugen, um einen Online-Validierungsdienst zur schnellen Depriorisierung mutmaßlich falscher Verifikationsergebnisse anzubieten und Verifikationszeugen graphisch darzustellen. Unser Vorschlag für ein Standardformat für Verifikationszeugen wurde inzwischen von mehr als dreißig verschiedenen Softwareverifikationswerkzeugen übernommen und unsere zeugenbasierten Validierungswerkzeuge sind zu einer Kernkomponente des Bewertungsschemas des Internationalen Softwareverifikationswettbewerbs geworden.

Acknowledgements

At a superficial level, this thesis appears to simply be the product of five years of my research. At a closer look, however, my work has been influenced by many people who I either had the privilege of working or living with, not only the last five years, but in some cases also long before. I am deeply grateful to all of them, and even though I cannot list all of them, I will try my best to mention the most influential ones.

I am grateful to the office administrators at our chair in Passau, Eva Veitweber and Eva Reichhart, and in Munich, Marianne Diem, to our technician, photographer, drone operator, and social-event manager Anton Fasching, to Prof. Dr. Rolf Hennicker, Prof. Dr. Mila Majster-Cederbaum, and Prof. Dr. Dr. h.c. Martin Wirsing, who, like Marianne Diem and Anton Fasching, could not have been more welcoming to all of us when our group moved from Passau to Munich, to our student research assistants in Passau, Alexander Driemeyer, Sebastian Ott, Nils Steinger, and Thomas Stieglmeier, and in Munich, Thomas Bunk, Alexander Koos, and Balthasar Schüß, and to my academic colleagues, Karlheinz Friedberger, who I shared an office with in Passau and whose office was right across the hallway from mine in Munich, which was very conducive for bouncing ideas off each other, Dr. Marie-Christine Jakobs, who has already begun extending my work and finding new applications of it in her own research, Nian-Ze Lee, who provided very valuable feedback on my most recent papers, Thomas Lemberger, who kept me company during many long and uncomfortable train rides between Passau and Munich, Martin Spießl, my successor in several scientific and administrative duties at the chair, Andreas Stahlbauer, who laid the groundwork for the implementation of verification witnesses, and Dr. Philipp Wendler, without whom CPAchecker would not be such a reliable and extensible basis for my research and that of many others. I am grateful to Prof. Dr. Stefan Leue and Prof. Dr. Andreas Podelski, who have kindly agreed to review this thesis. My supervisor and mentor, Prof. Dr. Dirk Beyer, deserves special emphasis. He has imprinted in me an urge to tenaciously strive for nothing short of perfection and leads by example, he always knew exactly which tasks to assign to me to ultimately further my academic progress, often long before I realized it, and he was always available to advise me on my research or my career.

Even despite of all the clerical and academic support I received, I must admit that I could not have sustained my efforts in completing this work, had I not had the tremendous moral support by my friends, who helped me escape reality every now and then, and my family: my sisters, with their inspiring strength and confidence, my mother, who kept reminding me that "no-one needs to be a doctor", my father, who had encouraged me to pursue computer science when I was an adolescent and who consequently always patiently listened to my technical and academic drivel, my magnificent and adorable bunnies Fritz, Frieda, Fred and Fonsi, and, most of all, my loving wife Vreni and my wonderful son Xaver, who always support me with their affection.

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass diese Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate und gedankliche Übernahmen kenntlich gemacht wurden.

Die Dissertation wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Passau, 9. März 2022	Matthias Dangl
Ort, Datum	Unterschrift

Contents

1. Introduction

The "internet of things" is often still portrayed as a very recent, sometimes even seemingly futuristic concept, where our refrigerators automatically order fresh groceries before we run out of them, where an artificial intelligence replaces human medical diagnosticians, and where the elderly or lonely of our society are supported by robots programmed to care for and socialize with them. While these examples are, in fact, not at all futuristic, but are all already available and being used, we do not need to look to such extremes to see that information technology in general has already and unquestionably pervaded our everyday lives. Consider the following questions:

- Do you own a smart phone?
- Do you use email?
- Do you use a messenger application?
- Do you use social media?
- Do you use a digital calendar, and if so, is it cloud-synchronized?
- How do you store and manage your family pictures?
- Do you track your fitness on a digital device, such as a smart watch or phone?
- Do you have a home entertainment system that allows you to stream any of hundreds of thousands of videos or songs?
- Do you use any digital household or garden appliances such as a vacuum-cleaning robot, a food processor, or robotic lawn mower?
- How many plastic cards do you carry in your wallet that contain RFID chips?
- Do you driver a car, or do you use public transport, such as bus, train, or metro?

All of these examples involve information technology, often to a much greater extent than most people expect [1], and probably, at least some of them apply to you, the reader. In today's world, we are all affected by information technology. The continuing rise of information technology since the latter half of the last century is changing our society in ways we are only beginning to understand. Whether we like this development and embrace the changes or despise, fear, and try to hide from this digital revolution: it is as irreversible as the industrial revolution, and we must not underestimate its impact on us.

Correctly working systems can entertain us, reduce our workload, and even save lives. Incorrectly working systems, however, can annoy us, cause delays, additional work and expense, and even harm or kill. The ever progressing automation of increasingly more complex tasks does not cease to re-assign jobs that only few years ago were considered to be far too complex for machines to handle, from humans to computers.

[1] The typical modern car contains about 50 microprocessors, and some advanced models may contain even up to three times that number. [52]

It must be pointed out that there are valid reasons to oppose the increasing pervasiveness of information technology. The automation of labor is an example of how even computational systems that work correctly according to specification can have highly controversial effects on our society: on the one hand, the process might ultimately make many lives easier, but on the other hand, in the mean time, many people may loose their income, fall into poverty, and never recover. Moreover, it can be argued that we may experience negative effects of the automation of labor and services even on the consumer side. There are controversies on whether computational systems will ever be able to replace the concept of human touch that many people have come to value, especially in areas of service that affect humans on a very personal level, such as health care and geriatrics [89]. It remains to be seen how resilient our society is to the effects of these changes. Other controversial examples are the application of information technology for governmental surveillance or military use cases. Objection to such applications may be motivated by civil rights or pacifist reasons, or by fear of abuse by bad actors or authoritarian regimes. However, even if we oppose a certain use of information technology that we cannot prevent, we should not hope for the involved computational systems to malfunction: Given a use of information technology we object to, the effects of a malfunction may be even more disastrous than we might imagine the system to be if it functions correctly, even in the hands of an ill-intentioned person or an opposing faction or organization, whether it is because an innocent person is misclassified as a criminal by a civil surveillance system or a rare lighting condition is misinterpreted by a missile-detection system during a time of pre-existing political tensions, as it happened in the Soviet nuclear false-alarm incident of 1983. Hence, whether we oppose or welcome the pervasiveness of information technology, our least common denominator should be that we strive for correctness.

Despite the need for correctness in our increasingly more complex computational systems, we already experience the side effects of this growing complexity: It becomes ever more difficult to comprehend, test, and maintain the systems we develop, and the severity of the consequences of system failures increases rapidly: Erroneous radiation-treatment equipment has killed multiple patients, self-driving cars have already been involved in several fatal accidents, and, as established above, the potential disaster that could be caused by malfunctioning military missile detection or guidance systems has long moved from fiction to being a real threat.

On the other hand, formal verification has already been established to be an effective countermeasure to these threats, from the verification of hardware-circuit designs, over the verification of flight-control systems, to the verification of operating-system drivers: The more critical a system is, or the more disastrous or expensive a failure is, the more likely we are to find formal verification in its development process. Consequently, there should be a high demand for automated formal software verification to cope with the large amounts of program code found in industrial settings.

However, despite the vast advancements made by the research community in the last decades regarding the effectiveness and efficiency of formal verification techniques, automated formal software verification has not yet crossed the line from a niche technology to a standard industry practice [2], which is made painfully obvious by the ubiquity of software errors we encounter in our daily lives.

1.1. Objectives

We identify two issues that contribute to the discrepancy between the academic success and the apparent lack of industrial adoption of software verification, formulate objectives to address these issues, and define and carry out tasks required to achieve these objectives.

1.1.1. Objective 1: Make Software Verification Applicable in Practice

Even for experts in the field, it is often difficult to decide which of the multitude of available techniques is the most suitable solution they should recommend to solve a given verification problem. How, then, should the average developer even begin to choose and apply a solution that fits their needs?

We improve the applicability of software verification to practical problems by defining and carrying out a set of tasks that address the issue of choosing the right solution for a given verification task. While completing these tasks, we contribute to the understanding of several popular algorithms for formal software verification. These contributions will help comprehend and further extend the described techniques, they provide an example and incentive for further comparative analysis of software-verification algorithms, and they constitute a baseline for automated solutions for determining the best verification technique for a given verification problem, thereby facilitating the practical application of formal verification in the industry.

Task: Investigate *k*-Induction

We first consider the possibility that there may already be an software-verification algorithm that is superior to all its alternatives: We pick a popular software-verification algorithm, k-induction, and investigate how it can be used effectively. We define and implement a framework for software verification with k-induction that allows us to express all existing approaches to k-induction in a uniform, module-based, and configurable architecture, and use it to test the following hypotheses:

- Existing variants of k-induction often use auxiliary invariants. An auxiliary-invariant generator is necessary to make k-induction successful in practice.
- The choice of invariant generator influences the effectiveness of k-induction.
- Our implementation of k-induction is more effective than all other existing implementations of k-induction that support the same input language if we configure it to use an invariant generator that produces weak invariants quickly and continuously refines its precision to supply stronger invariants as time progresses.
- k-induction is competitive with two other software-verification algorithms, namely predicate analysis and value analysis, if we configure it to use the same invariant generator as in the previous hypothesis.

While we found no evidence to support the suggestion that k-induction might be superior to its alternatives in general, the experiments we conduct confirm all of the above hypothesis [22].

Task: Identify Strengths and Weaknesses of *k*-Induction and Related Approaches

Building upon the insights gained from completing our first task, we continue our investigation into k-induction by comparing it to three other, closely related approaches, namely predicate abstraction, Impact, and bounded model checking, provide a conceptual unifying framework to highlight their similarities, differences, strengths, and weaknesses, give comprehensive examples of their application, and perform an extensive comparative experimental evaluation to test the following hypotheses:

- k-induction is consistently superior to the other three techniques, independent of the type verification task.
- Otherwise, we can identify the conceptual reasons that cause a given technique out of the four we investigate to be more or less successful than the other three for a given verification task.

Our experiments show that while overall, k-induction solves more verification tasks of our selected benchmark set than the other three approaches, we can identify subsets of tasks where k-induction is less successful than other approaches, i.e., we refute the hypothesis that k-induction is consistently superior. However, our experiments allow us to confirm the second hypothesis, i.e., we can identify and explain the conceptual reasons why one approach is more suitable than the others for a given verification task [24].

Task: Investigate PDR

We pick another popular algorithm for software verification, PDR, and investigate it. We present an example that illustrates a scenario where PDR is superior to plain k-induction, three data-flow analyses, and even k-induction combined with an invariant generator that comprises the three data-flow analyses. We then define and implement an extension of our framework for software verification with k-induction that allows us to perform PDR as a stand-alone verification algorithm, to combine k-induction and PDR into one algorithm that utilizes the PDR-aspect of guiding invariant discovery by leveraging failed induction attempts, and also to use this combination as an auxiliary-invariant generator to k-induction. Using this framework, we test the following hypotheses:

- Our extended framework is a suitable platform for implementing and evaluating PDR-based techniques.
- By providing k-induction with our new, PDR-based invariant generator we can improve the overall effectiveness of k-induction.
- On small programs, such as path programs, the new invariant generator is more effective than data-flow-based techniques and is therefore well-suited for the approach of generating path invariants [26].
- While the new PDR-based invariant generator is flexible, it is often outperformed by simpler, data-flow-based invariant-generation techniques.
- Our implementation is competitive when compared to the best available reference implementations.

Our experiments confirm the first, fourth, and fifth of these hypothesis, but not the second or third: We determine that on the chosen benchmark set, data-flow-based techniques are usually more efficient and, due to their efficiency, often also even more effective, because they do not exceed resource limitations as frequently. On the other hand, we are able to identify and explain relevant cases where our PDR-based approach was superior. By confirming that our framework is a suitable platform for PDR and that our implementation is competitive when compared to the available alternatives and also when compared to state-of-the-art implementations of other verification approaches, we support the argument that our conclusions are relevant and that the disadvantages are not caused by an inadequate implementation [15].

Task: Use Strategy Selection to Automate the Choice of Verification Technique

Because we know after completing the previous tasks that whether or not a given verification technique is successful depends on the type of verification task, we propose to use algorithm selection [93] to automatically choose from a given set of strategies the strategy that is most likely to succeed in solving a given verification task. We define four easy-to-determine Boolean features and implement a classifier that extracts from a given verification task a selection model based on these features and a configurable strategy selector. We use our implementation to test the following hypotheses:

- Given a set of strategies for software verification where we know for each strategy from our prior experience that it can be useful in practice, we can construct a sequential combination of these strategies that is more effective than each individual strategy.
- By classifying each verification task using our set of four Boolean features and selecting a strategy to solve a task from a set of three verification strategies based on this classification, we can further improve effectiveness significantly.
- The model-based selection strategy from the previous hypothesis is significantly more effective than ignoring the classification and randomly selecting strategies from the same set of strategies.

Our experiments confirm all three hypotheses. Consequently, we find that combinations of verification strategies are, in general, an effective solution to solve a heterogeneous set of verification task and that strategy selection in particular is a promising way to move the complexity of choosing the right verification strategy for a given task from the user of a verifier into the tool [13].

1.1.2. Objective 2: Make Software Verification Useful in Practice

Our first objective aims at making software verification applicable in practice, but even assuming a developer were able to apply some automated software verifier to their system, select a suitable verifier configuration, and obtain verification results, the developer would still require further guidance towards making practical use of these results. In the best case, verification results might be correct but still often differ vastly from the results developers obtain from classical quality-assurance techniques they are familiar with, such as testing.

In the worst case, differences between the assumptions made by developers about their system and the assumptions made by the verification tools may lead to verification results the developers will perceive as wildly inaccurate and useless. For example, a developer might spend hours to investigate a bug report produced by a verification tool, only to conclude that it is a false alarm. Consequently, developers may be tempted to discard formal verification as a waste of their own time and of computational resources that in their opinion could be better spent on testing.

We make software verification more useful in practice by defining and carrying out a set of tasks that address the issue of incomprehensible and incorrect verification results. The completion of these tasks will help unite formal verification with industrial practice as we define and implement a standard format for verification witnesses and use it to build tools and establish processes to make software verification and its results useful components of the software-development process.

Task: Establish a Standard Format for Verification Witnesses

We design, implement, and establish a machine-readable standard format for verification witnesses that can be used to store, compare, communicate, explain, visualize, and validate verification results. Verification witnesses constitute valuable artifacts of the verification process that can be used to preserve detailed information discovered during the analysis. We present two types of verification witnesses: violation witnesses, which represent violation results, i.e., found bugs, and correctness witnesses, which represent proof results. We describe verification witnesses as non-deterministic finite automata, formally define the process of consuming witness automata, illustrate their application using several comprehensive examples, and describe four different implementations of witness-based validators for violation results and two different implementations of witness-based validators for proof results. We use these implementations to test the following hypotheses:

- The standard format for verification witnesses is machine-readable and can be used to exchange verification witnesses between different verifiers.
- Violation-witness based validation of verification results can take considerably less effort than verification, i.e., the violation witness successfully guides the verifier through a considerably smaller state space.
- A high-precision violation-witness based result validator may improve the overall effectiveness of verification if an efficient but low-precision verifier produces witnesses and the validator rejects a substantial number of incorrect witnesses.
- The complexity of the validation of a correctness witness is related to the contents of the witness, i.e., there are verification tasks for which a verifier can produce witnesses such that the validation uses less resources to validate the witness than the verifier used to verify the verification task.

Our experiments confirm all four hypotheses and therefore demonstrate that verification witnesses are applicable to a heterogeneous set of verification tasks and to different verifiers, that verification witnesses can be used to increase the trust in verification results [18].

After publishing our findings, we expect other verifiers to adopt our format to confirm one more hypothesis:

- Our format for verification witnesses constitutes an established standard for the representation of verification results produced by software verifiers for the C programming language in the sense that the format is adopted by a wide and heterogeneous range of such software verifiers.

Because our format is already supported by over 30 different tools [8, 9], we can confirm the hypothesis and assert that it constitutes a tool-independent interface for verification results that helps avoid tool lock-in and provides a mechanism for increasing and establishing trust in the verification results of many different verifiers.

Task: Synthesize Test Cases from Verification Witnesses

We define and implement a process for synthesizing concrete (failing) test cases from violation witnesses. We use this implementation to test the following hypotheses:

- Concrete failing test cases can be synthesized from violation witnesses.
- By executing test cases synthesized from violation witnesses we can validate in verification results.

Our experiments confirm the hypotheses: We are able to use violation witnesses produced by a large number of different verifiers to synthesize concrete tests. By executing these tests, we can confirm verification results that we could not confirm using any other available witness-based verification-result validator. Consequently, we conclude that synthesizing tests from witnesses can bridge the gap between formal verification and standard development practice, because a developer can analyze such a failing test case using the tools they are already familiar with [21].

Task: Integrate Verification Witnesses into the Development Process

Lastly, we also define and implement an interactive web service to store, retrieve, validate and visualize verification witnesses. We illustrate how such a witness management system could become a useful component in the software-engineering process, where developers can use an integrated validation service to quickly de-prioritize suspected false alarms and generate graphical visualizations of witnesses that utilize verification information to aid the debugging process. We use the implementation of the web service to test the following hypothesis:

- Our toolchain is applicable to a large and established set of verification tasks and can be used to store, validate, and visualize verification witnesses.

We successfully inserted a large amount of witnesses produced by several different verifiers into the database of our toolchain, validated the witnesses, and generated reports with graphical visualizations, thereby confirming the hypothesis [12].

1.2. Structure

This thesis takes the form of a cumulative dissertation, i.e., it is a compilation of several individual manuscripts. In this chapter, we introduce the topic of the thesis, motivate our solutions, list our contributions, outline the structure of the thesis, give a high-level overview over the basic concepts of our research and describe our research method. In the second chapter, we list the titles of the manuscripts that comprise this thesis, discuss their interrelations and topical relevance, and attribute this author's share in their preparation and writing. In the fourth chapter, we summarize the contributions and discuss future prospects. Following the bibliography, we conclude this thesis with an appendix where we reprint each of the manuscripts without any changes, except that we enclose their pages with the headers and footers of this compilation to mark them with consecutive page numbers.

1.3. Background

In the following, we introduce the field of automated software verification and give a brief overview over recurring concepts that appear in our work. These concepts help us define the ideas we present in our manuscripts in a consistent and scientifically accurate formalism, so that our presentation is unambiguous and our contributions can be examined, built-upon, and compared with related work.

1.3.1. Automated Software Verification

Formal verification is the application of formal methods to construct a mathematical proof that a given system adheres to a given formal specification. *Software verification* is the application of formal verification to a software system, whereas *hardware verification* is the application of formal verification to a hardware system. Formal verification can be performed manually, semi-automatically, i.e., interactively using some kind of verification assistant, such as KeY [1], or fully automatic, i.e., by a verification tool. Considering the large amounts of program code that are found in today's real-world software systems, it can be argued that the only affordable way to scale software verification up to the industrial requirements, and while there is still value in manual and semi-automatic solutions, we consider only fully *automated software verification* in this thesis. In automated software verification, a *verification task* consists of the software *system* to be analyzed and the *specification* the system should adhere. In this thesis, we will only consider software systems where the source code is available, but solutions for verifying compiled executables, such as Java byte code, also exist [67]. A specification may require the verification of *safety properties*, such as whether some undesirable program location is unreachable or whether there is some invalid pointer dereference, it may require the verification of *liveness properties*, e.g., whether the program contains some infinite execution path (thereby exhibiting a behavior that is often informally described as "hanging" or "freezing"), or a combination of both safety and liveness properties. In this thesis, we mainly consider solutions based on reachability analysis that address safety properties, but techniques exist to adapt these solutions to liveness properties [96]. A verification task is processed by a software-verification tool, also called a *software verifier*. The software verifier attempts

to answer whether the system satisfies or violates the specification. Consequently, the answer produced by the verifier should either be TRUE, i.e., a proof was found that the system satisfies the specification, or FALSE, i.e., a bug was found that causes the system to violate the specification.

However, because the undecidable halting problem [100] can be reduced to solving the software verification problem, the problem of software verification is also undecidable in the general case. Therefore, a software verifier may not be able to solve a given verification task and instead give up, effectively answering UNKNOWN to the question whether the system satisfies the specification. Such a failure to solve a verification task may manifest itself as the verification process running indefinitely or exceeding a given time limit, exceeding the available memory resources, crashing, or simply giving up explicitly. In the general case, it is not even possible to determine if a failure could be avoided by allocating more computing resources.

While the fundamental issue of undecidability cannot be eliminated from the verification of infinite-state systems, such as software, techniques exist to mitigate it: Software verifiers may try to apply abstraction to a verification task, i.e., attempt to analyze it with a reduced precision and thereby considering an overapproximation of the state space, or they may compromise on the soundness of their analysis and consider only an underapproximation of the state space. A popular example of the use of abstraction is interval analysis [44]; a popular example of compromising on soundness is bounded model checking [31]. However, an imprecise software verifier may produce false alarms, i.e., its answer FALSE is not trustworthy. Conversely, an unsound software verifier may produce wrong proofs, i.e., its answer TRUE is not trustworthy. Again, techniques exist to counteract these effects. For example, a verifier may check its own results before reporting them to the user, and if it determines a problem, it can automatically refine its overapproximation [39] or its underapproximation [64] before re-attempting to solve the verification task. Nevertheless, due to the general undecidability of software verification, even these iterative refinement techniques cannot simultaneously guarantee the termination of the verification process within finite time and memory resources, and the correctness of the verification result. Moreover, in practice, there are additional, mundane, causes of incorrect verification results beyond scientific considerations: Due to the high complexity of many productively used programming languages, existing implementations of software verifiers sometimes lack the features to accurately model all aspects of a real-world system. In summary, we must conclude that if we want to apply software verification in practice, where infinite resources are not available, we will sometimes encounter inconclusive or incorrect results.

1.3.2. Configurable Program Analysis

In the field of software verification, it is often necessary to describe a program analysis. We use the formalism of configurable program analysis (CPA) [27, 40] for this purpose, which defines a conceptual framework to concisely express arbitrary types of program analyses. The CPA formalism follows the software-engineering principle of separation of concerns and enables modular compositions of reusable analysis components. It separates the definition of the abstract domain of a program analysis from the analysis algorithm.

We use this formalism as the basis of all program analyses and verification algorithms we research. This consistent use enables us to pinpoint the similarities and differences

between approaches, as we did for example in our comparison of the verification algorithms predicate abstraction, IMPACT, bounded model checking, and k-induction, where we were able to explain the algorithms and give detailed examples of each of them in one common formalism based on the CPA concept, and where we present an experimental evaluation of the algorithms using implementations in the verification framework CPACHECKER, which follow the formalism so closely that we are able to perform detailed measurements to attribute performance differences to individual aspects of each algorithm [24]. Such a detailed comparison would not be possible if each algorithm were only defined in a unique, monolithic formalism, because we could not so easily determine why one approach outperforms another on a given verification task and whether this difference is related to a unique aspect of one of the algorithms or to an independent, unrelated optimization.

We also use the CPA formalism to define how the information stored in a verification witness that is represented in the exchange format we propose can be extracted and associated with the intended program locations by a consumer [19]. Again, this consistent use makes it possible to examine the proposed technique not only in isolation but in a larger context of existing approaches to software verification.

1.3.3. Finite Automata

A *finite automaton*, also called finite-state machine, is a mathematical concept that is used to model computational systems. A finite automaton consists of at least the following components: a finite set of states Q a system can be in, an initial state $q_{INIT} \in Q$, a finite set of input symbols Σ also called alphabet, a transition relation $\delta \subseteq Q \times \Sigma \times Q$ that defines for each pair of states $q, q' \in Q$ and input symbol $\sigma \in \Sigma$ whether a transition from q to q' exists for the input symbol σ, and a set of accepting states $F \subseteq Q$. At any given time, a finite automaton is in exactly one of its states Q. Initially, this is the initial state q_{INIT}. The automaton switches from a state $q \in Q$ to a state $q' \in Q$ when an input symbol $\sigma \in \Sigma$ is read if $(q, \sigma, q') \in \delta$. If a finite automaton is used to distinguish between and a set of accepting states F. A finite automaton defines a language that contains a sequence $\langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ of $n \in \mathbb{N}$ input symbols (i.e., a word) if there is a corresponding sequence $\langle (q_0, \sigma_1, q_1), (q_1, \sigma_2, q_2), \ldots, (q_{n-1}, \sigma_n, q_n) \rangle$ of automaton transitions (i.e., $(q_{i-1}, \sigma_i, q_i) \in \delta$ with $1 \leq i \leq n$) where $q_0 = q_{INIT}$ and $q_n \in F$. The concept of *Büchi automata* extends finite automata to infinite inputs [35]: A Büchi automaton accepts only words that visit an accepting state $q \in F$ infinitely often.

Finite automata, their extension Büchi automata, and further variants of the concept of automata are a popular and recurring notion in the field of software verification [25, 38, 40, 51, 56, 68, 69, 70, 101]. Variants of finite automata that we use are *protocol automata*, which we introduce to model violation witnesses (verification witnesses for violations), and *observer automata*, which are a subtype of protocol automata that we use to model specifications and correctness witnesses (verification witnesses for proofs) [18].

1.3.4. Program Representation

All verification tasks we use in our examples and experiments consist of programs that are written in the programming language C. We choose the concept of *control-flow automata* (CFA) to model these programs. A control-flow automaton (L, l_{INIT}, G)

comprises a set L of program locations that represent the program counter, the program-entry location $l_{INIT} \in L$, and a set $G \subseteq L \times Ops \times L$ of control-flow edges. Each control-flow edge, also called CFA edge, represents the flow of control from one program location to another, during which an operation op of the set of program operations Ops is executed.

1.3.5. Error Paths

When a program is executed, the execution comprises a sequence of program operations. The specific sequence of operations that is executed is determined by external influences, also called the inputs to a program. We call a concrete sequence of such inputs a *test vector* [10].

A *program path* is a sequence $\langle (l_i, op_i, l_j), (l_j, op_j, l_k), \ldots, (l_m, op_m, l_n) \rangle$ of consecutive edges from the set G of control-flow edges that starts at the program-entry location, i.e., $l_i = l_{INIT}$. A program path is called *feasible* if a test vector exists for which the execution of the program corresponds precisely to the sequence of program operations on the path, otherwise the path is called *infeasible*.

If the sequence of control-flow edges of a program path and their corresponding operations violate a given specification, the program path represents a bug and is called an *error path*. A verification task, which consists of a program and a specification, is considered to be solved by verification if there is proof that no feasible error path exists, or by falsification if a feasible error path is found. Therefore, an error path can be said to constitute a *counterexample* to the claim that a program satisfies its specification. However, because the term "counterexample" is widely-used for related but different concepts in the general domain of model checking, we prefer to avoid confusion and use the term "error path" to describe a program path that violates a specification. The notion of error paths is an important concept in software verification, and just like there are efforts to identify root causes [42, 60, 63, 74, 80, 92, 107] in the domain of software engineering in general, the quality of the error reports produced by a software verifier is considered to be significant for the user experience [5]. Hence, there is a need for helping developers find the actual reason why an error path is feasible in their system [61, 73, 86, 95], for solutions to visualize errors and error paths [3, 4, 12, 62, 82], and for classifying bugs [7, 55, 90, 105].

In our work on documenting verification results as witnesses, we propose to represent an error path (or set of paths that contains error paths), i.e. a witness for a specification violation, as a violation witness [19]. These witnesses constitute a tool-independent machine readable interface for verification results upon which tools can be built to make the results more useful in practice and improve the user experience.

1.3.6. Test Generation

Testing is an important measure to improve software quality [87], and in industrial software projects there are often policies or even contractual agreements that specify certain requirements regarding software testing, for example minimum values for code-coverage metrics. It is also considered good practice to create test cases for known bugs in a software system, such that there is a way to determine whether a bug is fixed and a way to detect if the bug reappears at some later point to do further changes to the system. Test-driven development is a popular development method that proposes to create the test cases for a feature even before its implementation, to serve as a specification-by-example,

control implementation progress, and prevent regressions [6]. While even the successful execution of a large test suite cannot provide the same guarantees as a mathematical proof of safety produced by software verification, testing has certain advantages in practice: Tests are usually quick to execute, whereas proving that a program is correct is undecidable in the general case. Moreover, the execution and analysis of test cases does usually not require special training and is considered a standard task in software development.

Manually creating test cases that meet specific criteria, however, can be a time consuming task, which is why automatic test-case generation is a desirable goal, and why many techniques and tools for automatic test-case generation exist [36, 37, 41, 45, 57, 58, 59, 65, 66, 71, 72, 81, 83, 88, 91, 97, 102]. The approach of synthesizing test cases from error paths found by formal verification is more than fifteen years old [10, 102]. Since then, there have been efforts to go one step further to create debuggable executables with the intention to trigger a specification violation [85, 95], a recent one of which is our approach of designing a tool that converts error paths represented in our proposed format for verification witnesses into test harnesses, which can then be compiled and linked with the code under analysis, and executed to show the violation [21].

1.3.7. Inductive Invariants

A first-order formula or *assertion* that holds at every state of a system is called an *invariant* of that system. If we express a given specification as a first-order temporal-logic assertion, the task of verifying that a given program satisfies this specification is equivalent to checking that this assertion is an invariant of the program. Given two invariants P, P' where $P' \Rightarrow P$, we say that P' is *stronger* than P. A *location invariant* is an invariant that is restricted to a certain program location, i.e., a location invariant claims that an assertion always holds at a specific location, but it does not entail any assertions over other program locations. A *loop invariant* is a location invariant where the location is a loop head, i.e., the entry point into a loop [2]. A loop invariant is an assertion that must be true before and after every loop iteration, but this assertion need not be true at any other program location, including any location within the loop body, as long as it is restored at the end of the loop body.

We call a loop invariant *inductive* if we can prove it by induction, which entails checking two cases called the base case and the inductive-step case. In the base case, we need to prove that the assertion holds initially, i.e., before the first loop iteration. In the step case, we need to prove that every loop iteration preserves the assertion, or in other words, that if the assertion holds at the beginning of a loop iteration, it also holds after the loop iteration. For $k > 0$, we can generalize to concept of inductive invariants to k-inductive invariants, which are loop invariants that we can prove by k-induction. k-induction generalizes (1-)induction in the following way: In the base case, we need to prove that the assertion holds before the first k loop iterations, and in the step case, we need to prove that given any sequence of loop iterations with length k where the assertion holds before each iteration, the assertion must also hold after the k-th iteration. Every k-inductive invariant is also $(k+1)$-inductive. but the reverse is generally not true, which means that $(k+1)$-induction is more powerful than k-induction [104]. Given a

[2] Depending on the programming language, a loop may have more than entry point, but to keep our presentation simple, we only consider loops with precisely one entry point.

k-inductive invariant, however, it is possible to construct a stronger loop invariant that is (1-)inductive [75]. We use the fact that the power of k-induction increases with the value of k in our implementation of k-induction as a software-verification algorithm [22], where we also explain and give a detailed example of how not every invariant is k-inductive for any value of k.

To prove an invariant that is not k-inductive for any k, it is necessary to find a stronger invariant that is k-inductive for some $k > 0$. It is known that if P is an invariant of a program, an invariant P' must exist such that $P' \Rightarrow P$ and P' is an inductive invariant [84]. However, to show that a given k-inductive invariant is in fact k-inductive, we only need to check the base case and inductive-step case, which is usually much easier than finding such an invariant in the first place, which is an undecidable problem in general. We make use of this observation in our work on verification witnesses for proofs, where we argue that validating a given proof can be easier than computing the proof if the necessary inductive invariants are provided and need only be checked [16].

1.4. Research Method

Software verification is a branch of the field of software engineering. The computer scientist and software engineer Frederick Brooks claims that computer science is not an actual science because "the scientist builds in order to study; the engineer studies in order to build" [76]. He argues that software engineering differs from science in motivation and priorities, because in his opinion, the first priority of scientists should be the discovery of laws and facts itself and building tools should only be a means to achieve such discoveries, while engineers, on the other hand, would focus primarily on building tools with a concrete goal and perform research only as a means to build better tools.

Whether or not one agrees with the sentiment that the pursuit of concrete, practically applicable goals precludes the designation of "science" — we do not — the fundamental importance of scientific procedure is still indisputable, even in computer science: First, computer science is more than engineering, it is also closely tied to mathematics. Computer scientists create proofs, for example on the complexity or correctness of algorithms. In automated software verification, we create tools to automatically build mathematical models of software and apply formal correctness proofs to these models. While these proofs may sometimes be wrong — due to errors in the models or the proof construction — their formal mathematical nature makes them unambiguous and falsifiable. Therefore, a skeptic is able to inspect them, and, given enough time, detect and point out flaws. Second, in cases where mathematical proofs are not applicable, for example because an object of research is too complex to fully model it, a scientific claim still needs to be based on falsifiable statements and researchers must do their due diligence to test these statements and try to falsify them. If they fail to falsify a statement, i.e. their experiments "confirm" the claims, there is still no proof that the claims are correct, but again there is a means for skeptics who doubt the claims to falsify the statements themselves.

Software verification, as an engineering discipline, involves the development of concrete implementations of algorithms and techniques, as well as empirical studies of these implementations. We use experimental evaluations to investigate the advantages and disadvantages of a given technique as compared to its alternatives, or whether or not a proposed technique is practical at all. In the following, we describe how we conduct

empirical research, discuss the requirements we need to fulfill to enable the scientific process of testing falsifiable hypotheses, and give examples to show how we fulfill these requirements.

1.4.1. Hypothesis

Because our objectives are to make software verification more applicable and more useful in practice, our claims are often not easily amenable to proofs beyond fundamental aspects, like whether or not an algorithm is sound at all. Instead, we focus on an empirical research method, for example to show that one verification technique solves more verification tasks or is more efficient than another one under certain conditions. Consequently, we follow the scientific approach of supporting our claims with *falsifiable hypotheses* and experiments that test these hypotheses to build confidence in our conclusions. Thus, other researchers can take any of our hypotheses and test it. If their results support the hypothesis, it gains credibility; if they falsify it, we gain new knowledge and can devise a new hypothesis and design a better experiment. Either way, scientific advancement is made and there is neither necessity nor room for "alternative facts".

1.4.2. Experiment Design and Execution

To reach valuable scientific conclusions, it is necessary to consider threats to the experimental validity, design the experimental setup so that these threats are minimized, and document our considerations and the experimental setup. We distinguish between two aspects of experimental validity, namely external validity and internal validity.

External Validity

If we want to know if our conclusions are applicable to a given scenario, we must examine the *external validity* of our experimental setup, which defines the scope our conclusions are valid in. We must clarify under which conditions the experiments were conducted and consider whether or not these conditions are too restrictive to transfer the conclusions to the scenario in question. For example, all of our manuscripts describe experiments that are conducted using a particular benchmark suite of several thousands of verification tasks comprised of C programs[3]. This benchmark suite is established and widely-used in our research community, and it contains verification tasks extracted from real software systems, such as Linux device drivers. However, it is necessary to point out that if this composition of verification tasks is not representative of software-verification problems in general, we cannot generalize our conclusions. Likewise, for each of our experiments, it is our duty to state on which types of machines they were conducted, what the operating system was, what resource limitations we applied, and which versions of the tools and relevant libraries we used, because changing any of these variables could change the outcome of the experiment: While minor differences between version numbers of an operating system might not usually cause significantly different results, switching to a different operating system or tool implementation, or significantly reducing or increasing the amount of available resources is more likely to affect the results. One concrete example of how we address the concern for external validity is our work on witness-based result validation: When we

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

proposed the exchange format for verification witnesses, we implemented two validators in two different verification frameworks and showed not only the validation results of each validator for the witnesses produced by its own verification framework, but also the validation results for the witnesses of the respective other verification framework [19]. Had we only presented one validator and its validation results for witnesses produced with the same verification framework, we could not have believably claimed that the proposed format enables information exchange between verifiers. In fact, such a claim would have been wrong had we not discovered necessary improvements to our first draft of the format in the attempts to achieve cross-framework validation for our evaluation.

Internal Validity

If we want to know if our conclusions are actually supported by our experimental results within our concrete experimental setup, we must examine the *internal validity* of the setup, which requires us to consider and rule out alternative explanations that do not require our conclusions to be true, and to ensure that our experiments are *repeatable*, i.e., if we repeat a given experiment that we executed in a given setup and under a given set of conditions in the same setup and under the same conditions, we also obtain the same results within the margin of error expected for the assumed measurement precision. The internal validity of an experiment is threatened by confounding variables. We can improve the internal validity of our experiments by identifying and eliminating confounding variables or quantifying their effects. For example, while we would like to conduct our experiments quickly and might be tempted to optimize the "throughput" of our setup by executing multiple verification runs in parallel on a single machine, we must be aware that these processes may affect each other due to shared resources, such as caches or the bus. Thus, there is a trade-off between the throughput of our experimental setup and its internal validity. However, even if we always schedule only one verification run at a time on a given machine, there are still confounding variables. For example, if we allow the operating system to arbitrarily assign memory to processes on a machine with non-uniform memory access, a verification run that is assigned memory that is local to the run's set of processor cores may have an advantage over another run that is assigned memory that is farther away from the run's set of processor cores, which may significantly and nondeterministically influence our results [48]. These are just a few examples for confounding variables that affect experimental evaluations in software verification and therefore pose a threat to our conclusions. It is therefore important to always consider the internal validity of our experiments. To ensure reliable and accurate experiments for all our evaluations, we use the benchmarking framework BenchExec, which is based on extensive research on the requirements of reliable benchmarking in the context of software verification and which eliminates or at least mitigates many threats [29]. For example, BenchExec is able to enforce that all memory assigned to a given verification run is local to the run's set of processor cores.

1.4.3. Replicability

We call an experiment *replicable* if it can be repeated by another team of researchers that does not necessarily have access to the same (physical) laboratory equipment and machines, but can equally replicate the setup and conditions. Any claims made in

research that are based on experimental results must be replicable to be verifiable and therefore credible [33, 77, 79, 103]. Even though this principle should be common knowledge, experience reports and studies suggest that adherence to it is not self-evident [43, 94] for all researchers. Lack of replicability is also an impediment to research. For example, in our study on execution-based validation of verification results and the transformation of verification witnesses into concrete test cases [21], we could not include two of the verifiers we had intended to use, because they were unavailable to us due to their proprietary licenses.

To make our own software-verification experiments replicable, it is not only necessary to make each experiment repeatable, to ensure reliable results by performing accurate and precise measurements, and to document every experimental setup — requirements which we fulfill anyway in our considerations of experimental validity — but also to make all software and benchmarks that were use and all raw data obtained available. If the software or benchmarks we use is not available in the precise version stated in our research, other researchers cannot use it and therefore cannot replicate our experiments. Because academic publications usually restricted to a limited number of pages, experimental results are often condensed into plots and summarized in tables. However, if we do not also make the raw data available, this process of summarization may not be comprehensible and replicable for other researchers. Therefore, for each of our publications, we make a documented replication package available on a supplementary web page, together with any further supplementary material that may help readers benefit from our contributions. In addition to self-hosting our replication packages, which may be considered volatile, we also use a service provided by the European Organization for Nuclear Research (CERN) to archive them. [4] For example, the supplementary web page [5] for our work on violation witnesses [19] contains all raw data from the experiments presented in the paper, gives instructions on how to obtain the necessary software and benchmark set, and explains how to use the tools. Moreover, the supplementary documents the exchange format for witnesses presented in the paper and provides a virtual-machine image where the necessary prerequisites for running the experiments are already installed.

[4] https://zenodo.org/
[5] https://www.sosy-lab.org/research/verification-witnesses/

2. Discussion of Manuscripts

In this chapter, we state for each of the manuscripts that comprise this thesis the contributions of the thesis author and discuss, where applicable, their relation to other publications this author participated in.

2.1. Boosting *k*-Induction with Continuously-Refined Auxiliary Invariants

The article *Boosting k-Induction with Continuously-Refined Auxiliary Invariants*, which is reprinted in Appendix A, pages 52–70 of this dissertation, was authored by Dirk Beyer, Matthias Dangl, and Philipp Wendler, and published by Springer in the Proceedings of CAV 2015, pages 622–640 [22]. It corresponds to the task of investigating k-induction as part of our objective of making software verification applicable in practice, as outlined in Sect. 1.1.1.

The article describes an algorithm that combines k-induction for software verification with an invariant generator that refines its invariants over time. As a consequence, verification tasks for which weak invariants are sufficient for k-induction to succeed are solved quickly, without the need to unnecessarily spend time generating strong invariants, whereas such strong invariants may become available after some time for tasks for which k-induction requires them. The contributions of this article towards our objective of making software verification more applicable in practice are twofold: Not only do our contributions help better understand various variants of k-induction, we also provide a concrete approach for shifting the burden of choosing how much effort to spend on the computation of auxiliary invariants away from the user, who is likely incapable of making an informed decision, to the machine.

The definition of the algorithm based on the conceptual framework of configurable program analysis (see Sect. 1.3.2) and the implementation of the presented algorithm in the CPAchecker verification platform constitute elementary groundwork for the definition of a unifying conceptual framework and for the extensive comparative evaluation with other approaches that we presented in a later article [24]. The presented algorithm also served as a basis for our investigation of property-directed reachability, where we extend the k-induction algorithm to property-directed reachability and use it as a continuously refined invariant generator [15]. Both of these related articles are also part of this dissertation. A preprint of the article that provides some more details regarding the configurable abstract domain of the data-flow-based auxiliary-invariant generator used in the evaluation was published as a technical report [23].

Matthias Dangl is the main author of the article and responsible for more than 50 % of the article's contents. His contributions were (1) the conception and description of the presented algorithm with a focus on effectively combining invariant generation and k-induction, (2) the development of the presented tool implementation based on an

existing implementation of plain k-induction without auxiliary invariants, and (3) the execution and discussion of the experimental evaluation.

2.2. A Unifying View on SMT-Based Software Verification

The article *A Unifying View on SMT-Based Software Verification*, which is reprinted in Appendix A, pages 71–107 of this dissertation, was authored by Dirk Beyer, Matthias Dangl, and Philipp Wendler, and published by Springer in the Journal of Automated Reasoning (JAR) in March 2018, Volume 60, Issue 3, pages 299–335 [24]. It corresponds to the task of identifying strengths and weaknesses of k-induction and related approaches as part of our objective of making software verification applicable in practice, as outlined in Sect. 1.1.1.

The article presents a unifying conceptual framework for expressing SMT-based approaches to software verification, shows how to represent the four algorithms predicate abstraction, Impact, bounded model checking, and k-induction as instantiations of this framework, gives comprehensive examples for all presented concepts and algorithms, and provides an extensive and thorough experimental evaluation of the algorithms, followed by an analysis of the strengths and weaknesses of the compared approaches. Again, the contributions of this article towards our objective of making software verification more applicable in practice are twofold: First, the analysis of the strengths and weaknesses may be used directly to help decide which approach to use for a given verification problem. Second, the article's focus on a common formalism is supported by a running example that not only illustrates the formal concepts but may also serve to guide the reader by highlighting the connections between the concepts within the surrounding context of formal software verification. Using this textbook approach, we hope to help the reader deepen their understanding of the concepts of formal software verification and ultimately extend the framework to incorporate, analyze, and compare further approaches, to generate new knowledge on how to choose verification algorithms.

This article ties together, updates, and extends several earlier publications, in which the similarities between predicate abstraction and Impact were explored [30][1], bounded model checking and several variants of k-induction were compared and the general concept of k-induction with continuously refined invariant generation was introduced [22], and the four algorithms predicate abstraction, Impact, bounded model checking, and k-induction were evaluated and their strengths and weaknesses were analyzed [11]. One of the coauthors has published a dissertation that further extends the background, formal concepts and applications of the unifying framework [106]. Later publications build upon the presented conceptual framework and practical implementation by adapting k-induction to property-directed reachability [15] and by using the insights into strengths and weaknesses of the different approaches to automatically select an approach using algorithm selection [13].

Matthias Dangl is a coauthor of the article and responsible for more than 50 % of the article's contents. As main author of the two publications this journal article is based on [11, 22], his contributions were (1) his assistance in the definition of the presented algorithms as instantiations of the unifying conceptual framework, (2) the design and description of the extensive examples of all presented concepts and algorithms, and (3) the

[1] Not authored by the author of this dissertation

execution and discussion of the experimental evaluation, including the analysis of the strengths and weaknesses of the compared algorithms.

2.3. Software Verification with PDR: An Implementation of the State of the Art

The article *Software Verification with PDR: An Implementation of the State of the Art*, which is reprinted in Appendix A, pages 108–126 of this dissertation, was authored by Dirk Beyer and Matthias Dangl, and published by Springer in the Proceedings of TACAS 2018, pages 3–21 [15]. It corresponds to the task of investigating PDR as part of our objective of making software verification applicable in practice, as outlined in Sect. 1.1.1.

The article explores the effectiveness and efficiency of software verification with PDR. It presents an algorithm for property-directed k-induction that can be used as a standalone verification approach or as an invariant generator for k-induction. An extensive and thorough comparative experimental evaluation discusses strengths and weaknesses of the approach, and, combined with a reference implementation, serves as a baseline for ongoing research into software verification with PDR. These contributions aim to help better understand the advantages and disadvantages of PDR when compared to other approaches, and to facilitate making an informed choice of verification technique to use for a given task, which aligns with our objective of making software verification more applicable in practice.

An extended version of this article has been published as a technical report via arXiv in August 2019 with the eprint identifier 1908.06271 [14]. The presented algorithm is based on our earlier work on unifying SMT-based algorithms for software verification and is implemented in the same framework [24]. The approach of using a k-induction-based verification technique itself as an invariant generator for another k-induction procedure constitutes an extension of the KI↺KI approach we introduced in a previous publication and is an instance of the concept of continuously refined invariant generation for k-induction [22].

Matthias Dangl is the main author of the article and is responsible for more than 80 % of the article's contents. His contributions were (1) the definition and discussion of the presented algorithm, (2) the examples used to explain the presented core and background concepts, (3) the implementation of the presented algorithm and, for comparison, the CTIGAR algorithm [32] within the CPAchecker framework, and (4) the comparative experimental evaluation, including the analysis and discussion of its results.

2.4. Strategy Selection for Software Verification Based on Boolean Features

The article *Strategy Selection for Software Verification Based on Boolean Features*, which is reprinted in Appendix A, pages 127–142 of this dissertation, was authored by Dirk Beyer and Matthias Dangl, and published by Springer in the Proceedings of ISoLA 2018, pages 144–159 [13]. It corresponds to the task of using strategy selection to

automate the choice of verification technique as part of our objective of making software verification applicable in practice, as outlined in Sect. 1.1.1.

The article explains the concept of strategy selection for software verification, and identifies four Boolean features of verification tasks in the C programming language that can be statically determined and used to sufficiently distinguish between different input programs to a software verifier, such that a strategy selector can be defined that picks the verification algorithm based on the feature vector. Several verification strategies are discussed and a model-based strategy-selection function is defined. An experimental evaluation compares the verification strategies in isolation and as components of sequential combinations and feature-model-based strategy selection, showing that for the evaluated strategies, the defined model-based strategy-selection function yields results close to the theoretical optimum (i.e., an oracle that always chooses the best strategy), despite the simplicity of the chosen feature set. The article makes a strong case for further research on and implementation of model-based strategy selectors for software verifiers, because for software verification to become applicable in practice, as stated as our first objective, users should not be burdened by a — for them likely impossible — choice between verification techniques.

The idea of defining and implementing a model-based strategy selector logically follows our findings on the strengths and weaknesses of different verification techniques from our other publications [14, 22, 24]. One of the presented sequential combinations of verification strategies was taken from the 2018 International Competition on Software Verification (SV-COMP 2018) and an earlier version of it had already been presented in a publication that was coauthored by the author of this dissertation but is not part of this thesis [47].

Matthias Dangl is the main author of this article and is responsible for more than 80 % of the article's contents. His contributions were (1) the definition of the feature set, (2) the definition of the model-based selection function, (3) the choice, discussion and illustration of the presented verification strategies, (4) the implementation of the presented strategy selector in the CPAchecker verification framework, and (5) the comparative experimental evaluation, including the analysis and discussion of its results.

2.5. Verification Witnesses

The article *Verification Witnesses*, which is reprinted in Appendix A, pages 143–211 of this dissertation, was authored by Dirk Beyer, Matthias Dangl, Daniel Dietsch, Thomas Lemberger, Matthias Heizmann, and Michael Tautschnig, and published in the ACM journal Transactions on Software Engineering and Methodology (TOSEM) in November 2019 [18]. It corresponds to the task of establishing a standard format for verification witnesses as part of our objective of making software verification useful in practice, as outlined in Sect. 1.1.2.

The article presents the concept of verification witnesses and their application to validating verification results by allowing an independent validator to re-establish and thereby confirm the results of a software verifier. The article describes an exchange format that has been established as a standard for verification witnesses implemented by over 30 different verifiers and proposes to use this format to share verification information across tools, so that users can apply independent third-party tools to visualize, explore, and comprehend verification results. A technical specification of the format is provided and the conceptual

principles of verification witnesses, including their formal background, are discussed in detail. An expansive running example illustrates the formal concepts comprehensively and emphasizes the connections between the concepts within the surrounding context of witnesses for software verification. The paper concludes with an extensive experimental study on the application of witness-based result validation, using the validators CPAchecker, Ultimate Automizer, CPA-witness2test, and FShell-witness2test. The article constitutes an important contribution towards our objective of making software verification useful in practice, because verification witnesses are aimed at making verification results more accessible to engineers. Furthermore, the article also stays true to our objective of making software verification more applicable in practice, because instead of implementing an isolated and tool-specific solution, we focus on exchangeability and preventing tool lock-in, which makes it easier for a user to integrate the solution into their processes.

This journal article ties together and significantly extends our earlier work on violation witnesses [19] and correctness witnesses [16], both of which were published by ACM in the Proceedings of FSE 2015 and FSE 2016, respectively, and for both of which an extended abstract has also been published by the Gesellschaft für Informatik in SE 2016 [20] and SE 2017 [17], respectively. The article also overlaps with our work on execution-based witness validation [21], but focuses on the aspect of witness validation rather than test-case generation. Unlike our work on verification-aided debugging [12], this journal article focuses on the formalisms, conceptual foundations and general principles of verification witnesses, and their application to validating verification results, rather than on integrating verification results into the development process.

Matthias Dangl is the main author of this article and is responsible for more than 80 % of the article's contents. His contributions were (1) the consolidation of the conference papers the journal article is based on, (2) the discussion of the formal background, (3) the formalization of the concepts of verification witnesses and witness-based result validation, (4) the presentation of four violation-witness-based result validators and two correctness-witness-based result validators as applications of verification witnesses, (5) the technical specification of the exchange format for verification witnesses, (6) the running example, its illustration and its discussion, (7) the experimental evaluation, including the analysis of the evaluation results.

2.6. Tests from Witnesses

The article *Tests from Witnesses*, which is reprinted in Appendix A, pages 212–232 of this dissertation, was authored by Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig, and published by Springer in the Proceedings of TAP 2018, pages 3–23 [21]. It corresponds to the task of synthesizing test cases from verification witnesses as part of our objective of making software verification useful in practice, as outlined in Sect. 1.1.2.

The article introduces two new violation-witness based result validators, which, instead of performing model checking to re-establish the verification result, use the witness to construct a test harness and compile, link, and execute the verification-task program with the harness to check if the bug described by the witness can be observed in practice. The article emphasizes that in addition to the result validation, the generated test case itself constitutes a valuable artifact of the validation process that can be used to extend

the regression-test suite of the analyzed system. An extensive experimental evaluation during which more than 13 000 test cases were synthesized from verification witnesses supports the validity of the approach. The generation of test cases that trigger the bug reported by the verifier constitutes a significant contribution towards our objective of making software verification more useful in practice, because they make it possible for an engineer to reproduce and analyze the reported issue using the standard tools of their development environment that they are trained and experienced in, without requiring any prior knowledge on software verification or verification witnesses: The generated test cases are still valid in absence of the verification tool and are conceptually no different than any other test cases they already use.

This article continues earlier work on violation witnesses [19]. The description of the two additional violation-witness-based result validators, together with a reproduction of the experimental evaluation of witness-based result validation, was incorporated into the journal article on verification witnesses [18]. The aspect of test-case generation, on which *Tests from Witnesses* focuses, however, was out of scope for the journal article, which is why this article is reprinted separately in this dissertation.

Matthias Dangl is a coauthor of this publication and is responsible for more than 40 % of its contents. His contributions were (1) the definition and illustration of the approach of execution-based result validation with verification witnesses, (2) the conception and description of all examples, (3) the implementation of the violation-witness based result validator CPA-witness2test, (4) the conduction of preliminary experiments, and (5) his assistance in the analysis and discussion of the results of the experimental evaluation.

2.7. Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses

The article *Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses*, which is reprinted in Appendix A, pages 233–240 of this dissertation, was authored by Dirk Beyer and Matthias Dangl, and published by Springer in the Proceedings of CAV 2016, pages 502–509 [12]. It corresponds to the task of integrating verification witnesses into the development process as part of our objective of making software verification useful in practice, as outlined in Sect. 1.1.2.

The article explores further use cases of verification witnesses beyond result validation and test-case generation. It presents a development ecosystem in which software verification, witness-based result validation, and verification-witness storage and management are available as (potentially cloud-based) services, and where verification-witness-based interactive visualizations of verification results are available to developers as graphical verification reports. The article proposes to treat verification results — represented as verification witnesses — as valuable artifacts of the development processes, similar to how source code is managed in configuration-management systems, and how tasks and issues are managed in issue trackers, with cross references between the systems. Without such tight integration intro the development process, verification results will not be able to provide value for engineers. Conversely, the article's contributions therefore align directly with our objective of making software verification useful in practice.

The concepts presented in this article are based on the standardized exchange format for verification witnesses, and result validation is one of the use cases the article advocates [18]. The article complements our work on test-case generation [21] by adding interactive graphical reports to analyze witnesses and debug the issues they represent, by listing further use cases for verification witnesses, and by proposing an infrastructure that ties together all use cases in a common infrastructure to support the development process.

Matthias Dangl is the main author of this publication and responsible for more than 60 % of its contents. His contributions are (1) the implementation of the witness-management system and its integration into an existing verification infrastructure, (2) the description of the use cases, including witness collection, witness-based result validation, and verification-aided (i.e., witness-based) debugging using interactive graphical reports, (3) his assistance in the description of the proposed common infrastructure, and (4) the creation of the archive of collected examples of witnesses and interactive reports.

3. Summary and Prospects

To conclude this thesis, we summarize our work and contributions, emphasize the importance of our contributions regarding the objectives we defined in the introduction to this thesis, and discuss future prospects in alignment with our objectives.

3.1. Summary

Regarding our objective of making software verification more easily applicable, we have investigated several software-verification techniques and have designed and implemented a competitive approach to k-induction: By running a parallel combination of k-induction and a data-flow-based auxiliary-invariant generator that continuously refines its precision to produce stronger invariants as time progresses, we were able to create a verification approach that is able to successfully compete with other, established verification techniques, such as explicit-state model checking, predicate analysis, Impact, bounded model checking, and PDR [14, 22, 24]. We have contributed to the creation of a unifying conceptual framework to express the four SMT-based techniques predicate analysis, Impact, bounded model checking, and k-induction, and used this conceptual framework to present and explain comprehensive examples and to discuss the similarities and differences of the techniques. Furthermore, we have contributed to an implementation of the conceptual framework in the software verifier CPAchecker and used this implementation to conduct an extensive study to determine the benefits and drawbacks of each technique when applied to a large and heterogeneous set of verification tasks [24]. Because our findings indicate that the strengths of different techniques are often complementary, we have worked on combining these strengths to mitigate drawbacks of individual techniques. Consequently, we implemented an approach of algorithm selection, where a verifier automatically selects a verification strategy based on a feature vector extracted from a verification task, thereby removing the burden of choosing a suitable solution from the user [13]. We believe that these contributions are integral in making it easier for developers to *apply* software verification to their systems as part of their standard development process.

Regarding our objective of make software verification more useful in practice, we have devised and established a machine readable and tool-independent standard for storing and exchanging verification results: verification witnesses [18]. We have demonstrated that these witnesses can be used to validate [18], generate tests from [21], and visualize verification results [12]. Hence, verification witnesses help increase trust in verification results, explain them to developers, and can add substantial value to the development process. Because the format for verification witnesses is tool independent, a development process that uses witnesses as verification artifacts is not locked-in to one specific tool or vendor. For example users are free to swap out the verifier in their toolchain without affecting the components for validating or visualizing verification results, and vice versa. We therefore believe that our contributions are integral in making verification more *useful*

to developers as part of their everyday development practice and will in the long run help establish formal verification as a standard and best practice for quality assurance in professional software development, just like testing and continuous integration are already today.

3.2. Prospects

We have made significant contributions towards our objectives of making software verification more applicable and more useful in practice, but we cannot expect our efforts to suffice to make software verification a standard industry practice over night. Instead, we must critically reflect on our work and identify new, promising action items in alignment with our stated objectives. In the following, we will discuss several such action items that we believe to be some of the more obvious points of contact for further research, but certainly, many more ideas are conceivable.

3.2.1. Investigate Further Algorithms

We have investigated several popular algorithms for software verification, but considering the quick pace at which the research community develops new techniques and improvements to existing approaches, the study and comparative evaluation of these approaches as well as the design of a unifying theory are necessarily an ongoing and long-running effort. In the following, we will outline three further software-verification techniques that are closely related to those we discuss in this thesis, in that they are also SMT-based techniques and can be expressed similarly. Thus, we suggest extending our unifying framework to incorporate these techniques as a promising next step.

Automata-Based Trace Abstraction

Like predicate analysis and Impact, automata-based trace abstraction [68] uses counterexample-guided abstraction refinement. However, in automata-based trace abstraction, the iteratively refined abstract model of the program is not represented as a set of abstract states; instead an automaton is used to represent an overapproximation of the feasible program paths. If an infeasible error path is detected in this overapproximation, the abstract model is refined by computing interpolants for this path and using them to construct an automaton that represents a set of infeasible program paths (including the infeasible error path that caused the refinement), which is called a trace automaton. The refined abstract model is then obtained by intersecting the previous automaton with the complement of the trace automaton.

This approach can be considered to be similar to Impact in that a set of infeasible paths obtained by generalizing an infeasible error path by using interpolation is directly removed from the abstract model, as opposed to predicate abstraction, which uses interpolants to first refine its abstract domain and thereby indirectly refines its abstract model. It would therefore be interesting to investigate the following three variants of trace abstraction and compare them to each other and to the related techniques we already investigated: first, a direct implementation of automata-based trace abstraction, i.e., using a data structure specifically designed to represent automata as an abstract model as discussed

above; second, explicitly encoding the automata that represent the abstract model and the trace abstraction as SMT formulas, which can then be incorporated into our unifying framework; third, Impact.

Slicing Abstractions

Another technique that employs counterexample-guided abstraction refinement is the approach of slicing abstractions [34, 53], also called "state splitting". The idea of slicing abstractions is to first construct an abstract-reachability graph where every abstract state is labeled *true*. Then, the technique iterates over the following three steps: First, the algorithm searches an infeasible error path in the graph. Second, it computes interpolants for this path. Third, it refines the abstraction by splitting (i.e., duplicating) each abstract state (and its subgraph) where one represents a part of the state space that intersects with the interpolant and the other one represents a part of the state space that intersects with the negation of that interpolant. All outgoing branches of the new states are then immediately checked for feasibility to guarantee that ultimately, the infeasible error path is removed from the abstract-reachability graph and will not be encountered again.

This approach of slicing abstractions is comparable to Impact in that it directly conjoins the interpolant to a state — as opposed to using it to refine the abstract domain like in predicate abstraction — and has been implemented in the tools Slab [50], Ultimate Kojak [54], and recently also in CPAchecker [98]. The similarity to Impact and the apparent conceptual similarities to the techniques we already investigated [24] make slicing abstractions an interesting candidate for further research into extending our unifying framework and conducting further comparative evaluations.

Symbolic Execution

Symbolic execution [78] is a technique that explores each program path separately and interprets its operations. The abstract states in a symbolic-execution analysis consist of two parts: (1) a symbolic store that tracks symbolic and concrete values of program variables and (2) a set of constraints over the symbolic values. Each time a variable is assigned a nondeterministic value, a fresh symbolic value is created and mapped to the variable in the symbolic store. If, on the other hand, a concrete value for a variable can be unambiguously determined by the analysis, then this concrete value is mapped to the variable in the store. Assumptions over program variables, such as those imposed by the conditions of branching statements encountered along a path, are tracked as constraints over symbolic values. Whenever the feasibility of a path needs to be determined, for example if an error location is encountered, the constraints are checked for satisfiability, using the symbolic store as interpretation. We have already discussed in this thesis how our existing unifying framework can be configured as an analysis that behaves similarly to symbolic execution [24] by applying the CPA algorithm and configuring the Predicate CPA to use the merge operator merge^{sep} and the block operator blk^{never}. The purpose of using the operator merge^{sep} instead of $\mathsf{merge}_{\mathbb{P}}$ is to build a reachability tree by preventing all merges between abstract states and therefore keeping all paths separate. Like in bounded model checking, the purpose of using the operator blk^{never} is to disable abstraction computations; consequently, the semantics of all program operations of a path are accumulated in the path formula of abstract states during the state-space exploration.

Effectively, this results in an analysis similar to bounded model checking, except that here, feasibility is checked as soon as a specification violation is detected on a path as opposed to checking the feasibility of all violating paths after a certain amount of loop iterations. One difference between this approach and the literature definition of symbolic execution is that this approach tracks all values in the path formula instead of separating them from the constraints in a dedicated symbolic store, which may affect the performance. However, it is conceivable that this overhead might by negligible, because propagating constants is likely just as simple for the SMT solver as it is for the program analysis. Furthermore, as a benefit of our configurable framework, we can consider creating a new variant of symbolic execution by switching the merge operator from merge^{sep} to $\mathsf{merge}_{\mathbb{P}}$ to mitigate the issue of state-space explosion, because the operator $\mathsf{merge}_{\mathbb{P}}$ allows us to join two abstract states of separate paths into a one common abstract state that precisely represents the disjunction of both paths.

3.2.2. Extend Algorithm Selection

In our work on algorithm selection, we point out that while our results emphasize the promising prospects of applying algorithm selection to software verification and provide a baseline for further empirical research in this direction, there are still several important aspects that need to be addressed in future research [13].

First, our evaluation is constrained to a limited set of verification strategies. As such, it is dependent on the strategies in this limited image range that the strategy selector maps to, whereas extending the range to further verification strategies and different implementations could further improve effectiveness and general versatility.

Second, we only evaluated our selection model on a given benchmark set. While this benchmark set is taken from the largest and most diverse publicly available set of verification tasks, we have no guarantee that our model is useful for selecting verification strategies for verification tasks outside this repository. In fact, considering that the features that comprise our selection model only express whether a certain feature of the programming language is present or not in a verification task, it is even likely that most sufficiently large verification tasks from real-world applications would not be distinguishable from each other using our selection model. Consequently, it will be necessary to refine the selection model. For example, instead of only modeling whether or not a verification task contains floating-point variables, the model might expose the ratio of the number of floating-point variables to the total number of variables in the task.

Third, we considered only one verification property in the design of our strategy selector and the selection of the benchmark set because within CPAchecker, which is the framework we implemented our approach in, the variety of verification strategies to choose from is too limited to be interesting for other properties. In practice, however, CPAchecker can be configured to use the verification property as part of the selection model and this option is essential for properly and effectively handling verification tasks with different properties: For example, checking whether a program terminates requires a different approach than checking whether a given program location is reachable. and in the strategy selector. For benchmark sets with more than one verification property, it is therefore important to define a strategy selector that considers the verification property as an additional feature to distinguish between tasks.

Fourth, we optimized our strategy selector for a given definition of quality, namely the the scoring schema from the International Competition on Software Verification (SV-COMP) [9]. While this scoring schema follows a community consensus that values safety higher than finding bugs and punishes wrong answers severely, not every user can be expected to agree with this schema. For example, some users might prefer a pragmatic approach that focuses on finding as many bugs as possible instead of striving for absolute safety, which may be an unattainable goal in many large industrial software projects.

Fifth, our approach is based solely on verification expert knowledge some experimentation to fine-tune the strategy selector. This was feasible for us due to the limitations to a small set of simple features, a small set of verification strategies, and a known set of verification tasks. To achieve the improvements outlined in this section and make algorithm selection more versatile, it may be beneficial to apply techniques from the domain of machine learning. Preliminary research in this direction already exists in the domain of software verification [46, 49, 99] and combining these techniques with our knowledge on the strengths and weaknesses of different verification strategies is an interesting topic for further research that is likely to yield useful tools for making software verification easier to apply in practice.

3.2.3. Integrate Verification into IDEs

Many large software projects today are developed using integrated development environments (IDEs) like for example Eclipse [1], IntelliJ IDEA [2], or Visual Studio [3]. These development environments provide, among other features, not only the ability to write and display program code, but also to auto-complete code, guide developers through APIs refactor software designs, generate test cases, build, run and debug systems, and even perform static analysis to identify potential bugs. Advantages of integrating these various aspects of software development into one common environment are that developers do not need to switch between tools, which would unnecessarily waste time and distract them from their tasks, and that the common user interface facilitates the comprehensibility of less well-known features by enabling users to visually (and sometimes also linguistically) relate them directly to features they already know well.

Consequently, adding support for software verification to IDEs might be a means to expedite practical adoption of software verification. For the Eclipse IDE, there already exists a plug-in module to integrate the CPROVER model-checking framework, which includes the bounded model checker Cbmc. [4] It would be an interesting research topic in the field of empirical software engineering to confirm or disprove the claim that such an IDE feature is actually useful to developers in practice.

3.2.4. Build More Tooling around Witnesses

We have established a common format for verification witnesses that is already supported by more than 30 software verifiers, and we have built several different result validators [18] and one visualization tool based on these witnesses [12]. However, to further increase

[1] https://www.eclipse.org
[2] https://www.jetbrains.com/idea
[3] https://visualstudio.microsoft.com
[4] https://www.cprover.org/eclipse-plugin/

adoption of verification witnesses, support our claims of the ubiquity and tool-independence of the format, and make witnesses more useful in practice, we need to not only improve existing tools but also to create new applications and new tools for existing applications of verification witnesses.

For example, we claim that untrusted verification results can be validated using witness-based result validators. However, while many software verifiers are able to produce verification witnesses, only four validators for violation witnesses and two validators for correctness witnesses exist to date, and the set of features supported by these validators is necessarily limited to the features provided by the frameworks they are embedded in. Instead, it would be desirable to be able to apply any software verifier as a witness validator.

A similar problem existed until recently for the concept of conditional model checking [25]. In conditional model checking, a model checker that is unable to completely solve a verification task does not simply give up with the answer UNKNOWN but instead outputs a condition that describes which parts of the state space it successfully checked before giving up. A second model checker, called the conditional model checker, could then attempt to check only the remaining parts of the program. Consequently, there may be tasks that neither of the two tools could solve on its own but that can be solved by their combination. In practice, however, this concept did not gain traction, because to combine two such model checkers in this way, both would need to agree on a common representation and understanding of the condition that is passed from the first tool to the second one, and the second tool would need to be able to efficiently apply this condition within its verification approach. A recent solution to the second part of this dilemma is to create a *reducer* that can parse a condition and encode it directly into the input program using the programming language itself, i.e. reduce the input program to the parts of the state space not yet checked by the first tool [28]. Thus, a conditional model checker can be constructed from any off-the-shelf model checker by combining it with a reducer.

This idea is transferable to witness-based result validation: Just like a reducer can be used to encode a given condition into a given input program to enable conditional model checking, we could also build a tool to encode a witness automaton into an input program that can then be verified by any off-the-shelf software verifier, thereby validating the verification result described by the witness. For violation witnesses, this is basically the same operation that is performed by the reducer in conditional model checking, because a violation witness simply restricts the program state space, just like the condition in conditional model checking. Note that the construction of a test harness performed by dynamic witness-based result validators is simply a special case of this operation, because it restricts the program state space to precisely one path. For correctness witnesses, encoding the witness automaton into the program may not be as straight-forward as for violation witnesses, but one idea would be to use the invariants from the witness as assertions in the program to force the verifier to check the invariant.

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The key tool. *Software and System Modeling*, 4(1):32–54, 2005.

[2] J. Alglave, A. F. Donaldson, D. Kröning, and M. Tautschnig. Making software verification tools really work. In *Proc. ATVA*, LNCS 6996, pages 28–42. Springer, 2011.

[3] H. Aljazzar and S. Leue. Debugging of dependability models using interactive visualization of counterexamples. In *Proc. QEST'08*, pages 189–198. IEEE, 2008.

[4] C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *Proc. COMPSAC*, pages 541–546. IEEE, 2007.

[5] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proc. POPL*, pages 97–105. ACM, 2003.

[6] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[7] M. T. Befrouei, C. Wang, and G. Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. *FMSD*, 49(1-2):1–32, 2016.

[8] D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In *Proc. TACAS*, LNCS 9636, pages 887–904. Springer, 2016.

[9] D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017.

[10] 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.

[11] D. Beyer and M. Dangl. SMT-based software model checking: An experimental comparison of four algorithms. In *Proc. VSTTE*, LNCS 9971, pages 181–198. Springer, 2016.

[12] D. Beyer and M. Dangl. Verification-aided debugging: An interactive web-service for exploring error witnesses. In *Proc. CAV (2)*, LNCS 9780, pages 502–509. Springer, 2016.

[13] D. Beyer and M. Dangl. Strategy selection for software verification based on boolean features: A simple but effective approach. In *Proc. ISoLA*, LNCS 11245, pages 144–159. Springer, 2018.

[14] D. Beyer and M. Dangl. Software verification with PDR: Implementation and empirical evaluation of the state of the art. *arXiv/CoRR*, 1908(06271), August 2019. `http://arxiv.org/abs/1908.06271`.

[15] D. Beyer and M. Dangl. Software verification with PDR: An implementation of the state of the art. In *Proc. TACAS (1)*, LNCS 12078, pages 3–21. Springer, 2020.

[16] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proc. FSE*, pages 326–337. ACM, 2016.

[17] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Exchanging verification witnesses between verifiers. In *Proc. SE'17*, page 93. Gesellschaft für Informatik, 2017.

[18] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4):57:1–57:69, 2022.

[19] 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.

[20] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Verification witnesses. In *Proc. SE'16*, pages 105–106. Gesellschaft für Informatik, 2016.

[21] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. Tests from witnesses: Execution-based validation of verification results. In *Proc. TAP*, LNCS 10889, pages 3–23. Springer, 2018.

[22] D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. CAV*, LNCS 9206, pages 622–640. Springer, 2015.

[23] D. Beyer, M. Dangl, and P. Wendler. Combining k-induction with continuously-refined invariants. Technical Report MIP-1503, University of Passau, January 2015. arXiv:1502.00096.

[24] D. Beyer, M. Dangl, and P. Wendler. A unifying view on SMT-based software verification. *J. Autom. Reasoning*, 60(3):299–335, 2018.

[25] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proc. FSE*. ACM, 2012.

[26] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM, 2007.

[27] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.

[28] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. Reducer-based construction of conditional verifiers. In *Proc. ICSE*, pages 1182–1193. ACM, 2018.

[29] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer*, 21(1):1–29, 2019.

[30] D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In *Proc. FMCAD*, pages 106–113. FMCAD, 2012.

[31] 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.

[32] J. Birgmeier, A. R. Bradley, and G. Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Proc. CAV*, LNCS 8559, pages 831–848. Springer, 2014.

[33] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. Replication's role in software engineering. In *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer, 2008.

[34] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundam. Inform.*, 89(4):369–392, 2008.

[35] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. ICLMPS*, pages 1–11. Stanford University Press, 1962.

[36] 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.

[37] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proc. CCS*, pages 322–335. ACM, 2006.

[38] M. Chapman, H. Chockler, P. Kesseli, D. Kröning, O. Strichman, and M. Tautschnig. Learning the language of error. In *Proc. ATVA*, LNCS 9364, pages 114–130. Springer, 2015.

[39] 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.

[40] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*. Springer, 2018.

[41] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[42] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. ICSE*, pages 342–351. ACM, 2005.

[43] C. S. Collberg and T. A. Proebsting. Repeatability in computer-systems research. *Commun. ACM*, 59(3):62–69, 2016.

[44] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. Int. Symp. on Programming*, pages 106–130. Dunod, 1976.

[45] C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. ICSE*, pages 422–431. ACM, 2005.

[46] M. Czech, E. Hüllermeier, M. Jakobs, and H. Wehrheim. Predicting rankings of software verification tools. In *Proc. SWAN*, pages 23–26. ACM, 2017.

[47] M. Dangl, S. Löwe, and P. Wendler. CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 423–425. Springer, 2015.

[48] A. B. de Oliveira, J.-C. Petkovich, and S. Fischmeister. How much does memory layout impact performance? A wide study. In *Proc. REPRODUCE*, 2014.

[49] Y. Demyanova, H. Veith, and F. Zuleger. On the concept of variable roles and its use in software analysis. In *Proc. FMCAD*, pages 226–230. IEEE, 2013.

[50] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. Slab: A certifying model checker for infinite-state concurrent systems. In *Proc. TACAS*, LNCS 6015, pages 271–274. Springer, 2010.

[51] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proc. SPIN*, pages 57–79, 2001.

[52] P. A. Eisenstein. Alexa start my car: How next year's models are in the fast lane for high tech. *NBC News*, 2017.

[53] E. Ermis, J. Hoenicke, and A. Podelski. Splitting via interpolants. In *Proc. VMCAI*, LNCS 7148, pages 186–201. Springer, 2012.

[54] E. Ermis, A. Nutz, D. Dietsch, J. Hoenicke, and A. Podelski. Ultimate Kojak (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 421–423. Springer, 2014.

[55] E. Ermis, M. Schäf, and T. Wies. Error invariants. In *Proc. FM*, LNCS 7436, pages 187–201. Springer, 2012.

[56] A. Farzan, M. Heizmann, J. Hoenicke, Z. Kincaid, and A. Podelski. Automated program verification. In *Proc. LATA*, LNCS 8977, pages 25–46. Springer, 2015.

[57] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.

[58] P. Godefroid and K. Sen. Combining model checking and testing. In *Handbook of Model Checking*, pages 613–649. Springer, 2018.

[59] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. ISSTA*, pages 53–62. ACM, 1998.

[60] A. Groce, S. Chaki, D. Kröning, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.

[61] A. Groce and D. Kröning. Making the most of bmc counterexamples. *Electr. Notes Theor. Comput. Sci.*, 119(2):67–81, 2005.

[62] A. Groce, D. Kröning, and F. Lerda. Understanding counterexamples with explain. In *Proc. CAV'04*, LNCS 3114, pages 453–456. Springer, 2004.

[63] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Proc. SPIN*, LNCS 2648, pages 121–135. Springer, 2003.

[64] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Proc. POPL*, pages 122–131. ACM, 2005.

[65] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proc. FSE*, pages 117–127. ACM, 2006.

[66] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proc. ASE*, pages 219–228. IEEE, 2000.

[67] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. J. Softw. Tools Technol. Transfer*, 2(4):366–381, 2000.

[68] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *Proc. SAS*, LNCS 5673, pages 69–85. Springer, 2009.

[69] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *Proc. POPL*, pages 471–482. ACM, 2010.

[70] 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.

[71] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. How did you specify your test suite. In *Proc. ASE*, pages 407–416. ACM, 2010.

[72] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and infeasible path analysis. In *Proc. ISSTA*, pages 95–107. ACM, 1994.

[73] M. Jose and R. Majumdar. Bug-assist: Assisting fault localization in ANSI-C programs. In *Proc. CAV*, LNCS 6806, pages 504–509. Springer, 2011.

[74] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proc. PLDI*, pages 437–446. ACM, 2011.

[75] D. Jovanovic and B. Dutertre. Property-directed k-induction. In *Proc. FMCAD*, pages 85–92. IEEE, 2016.

[76] F. P. B. Jr. The computer scientist as a toolsmith ii. *Commun. ACM*, 39(3):61–68, 1996.

[77] N. Juristo and O. S. Gómez. Replication of software engineering experiments. In *Empirical Software Engineering and Verification*, pages 60–88. Springer, 2012.

[78] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[79] S. Krishnamurthi and J. Vitek. The real software crisis: Repeatability as a core value. *Commun. ACM*, 58(3):34–36, 2015.

[80] S. Leue and M. T. Befrouei. Counterexample explanation by anomaly detection. In *Proc. SPIN*, LNCS 7385, pages 24–42. Springer, 2012.

[81] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. Residual investigation: predictive and precise bug detection. In *Proc. ISSTA*, pages 298–308. ACM, 2012.

[82] G. Maheswara, J. S. Bradbury, and C. Collins. TIE: an interactive visualization of thread interleavings. In *Proc. SoftVis*, pages 215–216, 2010.

[83] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proc. ICSE*, pages 416–426. IEEE, 2007.

[84] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety.* Springer, 1995.

[85] P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *Proc. FM*, LNCS 6664, pages 73–87. Springer, 2011.

[86] V. Murali, N. Sinha, E. Torlak, and S. Chandra. What gives? A hybrid algorithm for error trace explanation. In *Proc. VSTTE*, pages 270–286, 2014.

[87] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing.* Wiley Publishing, 3rd edition, 2011.

[88] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *Proc. ICSM*, pages 1–10. IEEE, 2010.

[89] J. Parviainen, T. Turja, and L. v. Aerschot. Robots and human touch in care: Desirable and non-desirable robot assistance. In *Proc. ICSR'18*, LNCS 11357, pages 533–540. Springer, 2018.

[90] A. Podelski, M. Schäf, and T. Wies. Classifying bugs with interpolants. In *Proc. TAP*, LNCS 9762, pages 151–168. Springer, 2016.

[91] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Softw. Eng.*, 2(4):293–300, 1976.

[92] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. ASE*, pages 30–39. IEEE, 2003.

[93] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[94] E. F. Rizzi, S. Elbaum, and M. B. Dwyer. On the techniques we create, the tools we build, and their misalignments: A study of KLEE. In *Proc. ICSE*, pages 132–143. ACM, 2016.

[95] H. Rocha, R. S. Barreto, L. C. Cordeiro, and A. D. Neto. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *Proc. IFM*, LNCS 7321, pages 128–142. Springer, 2012.

[96] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006.

[97] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. FSE*, pages 263–272. ACM, 2005.

[98] M. Spiessl. Configurable software verification based on slicing abstractions. Master's Thesis, LMU Munich, Software Systems Lab, 2018.

[99] V. Tulsian, A. Kanade, R. Kumar, A. Lal, and A. V. Nori. MUX: Algorithm selection for software model checkers. In *Proc. MSR*. ACM, 2014.

[100] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proc. LMS*, volume s2-42, pages 230–265. London Mathematical Society, 1937.

[101] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata (Proc. Banff'95)*, LNCS 1043, pages 238–266. Springer, 1996.

[102] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. ISSTA*, pages 97–107. ACM, 2004.

[103] J. Vitek and T. Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proc. EMSOFT*, pages 33–38. ACM, 2011.

[104] T. Wahl. The k-induction principle, 2013. Available at `http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf`.

[105] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. ICSE*, pages 461–470, 2008.

[106] P. Wendler. Towards practical predicate analysis. PhD Thesis, University of Passau, Software Systems Lab, 2017.

[107] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. FSE*, pages 1–10. ACM, 2002.

A. Manuscripts

In this chapter, we reprint the manuscripts we wrote with the goal to accomplish the tasks we defined to reach the objectives of this thesis described in Chapter 1. Each manuscript corresponds to exactly one task. The order in which the manuscripts appear in the following corresponds to the order of the objectives and corresponding tasks from Chapter 1, i.e., the manuscripts are ordered and grouped semantically rather than chronologically in the order of their publication dates.

Boosting *k*-Induction with Continuously-Refined Invariants

Dirk Beyer, Matthias Dangl, and Philipp Wendler

University of Passau, Passau, Germany

Abstract. *k*-induction is a promising technique to extend bounded model checking from falsification to verification. In software verification, *k*-induction works only if auxiliary invariants are used to strengthen the induction hypothesis. The problem that we address is to generate such invariants (1) automatically without user-interaction, (2) efficiently such that little verification time is spent on the invariant generation, and (3) that are sufficiently strong for a *k*-induction proof. We boost the *k*-induction approach to significantly increase effectiveness and efficiency in the following way: We start in parallel to *k*-induction a data-flow-based invariant generator that supports dynamic precision adjustment and refine the precision of the invariant generator continuously during the analysis, such that the invariants become increasingly stronger. The *k*-induction engine is extended such that the invariants from the invariant generator are injected in each iteration to strengthen the hypothesis. The new method solves the above-mentioned problem because it (1) automatically chooses an invariant by step-wise refinement, (2) starts always with a lightweight invariant generation that is computationally inexpensive, and (3) refines the invariant precision more and more to inject stronger and stronger invariants into the induction system. We present and evaluate an implementation of our approach, as well as all other existing approaches, in the open-source verification-framework CPAchecker. Our experiments show that combining *k*-induction with continuously-refined invariants significantly increases effectiveness and efficiency, and outperforms all existing implementations of *k*-induction-based verification of C programs in terms of successful results.

1 Introduction

Advances in software verification in recent years have lead to increased efforts towards applying formal verification methods to industrial software, in particular operating-systems code [3,4,34]. One model-checking technique that is implemented by half of the verifiers that participated in the 2015 Competition on Software Verification [7] is bounded model checking (BMC) [16,17,22]. For unbounded systems, BMC can be used only for falsification, not for verification [15]. This limitation to falsification can be overcome by combining BMC

A preliminary version of this article appeared as technical report [8].

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 622–640, 2015.
DOI: 10.1007/978-3-319-21690-4_42

with mathematical induction and thus extending it to verification [26]. Unfortunately, inductive approaches are not always powerful enough to prove the required verification conditions, because not all program invariants are inductive [2]. Using the more general k-induction [38] instead of standard induction is more powerful [37] and has already been implemented in the DMA-race analysis tool Scratch [27] and in the software verifier Esbmc [35]. Nevertheless, additional supportive measures are often required to guide k-induction and take advantage of its full potential [25]. Our goal is to provide a powerful and competitive approach for reliable, general-purpose software verification based on BMC and k-induction, implemented in a state-of-the-art software-verification framework.

Our contribution is a new combination of k-induction-based model checking with automatically-generated continuously-refined invariants that are used to strengthen the induction hypothesis, which increases the effectiveness and efficiency of the approach. BMC and k-induction are combined in an algorithm that iteratively increments the induction parameter k (iterative deepening). The invariant generation runs in parallel to the k-induction proof construction, starting with relatively weak (but inexpensive to compute) invariants, and increasing the strength of the invariants over time as long as the analysis continues. The k-induction-based proof construction adopts the currently known set of invariants in every new proof attempt. This approach can verify easy problems quickly (with a small initial k and weak invariants), and is able to verify complex problems by increasing the effort (by incrementing k and searching for stronger invariants). Thus, it is both efficient and effective. In contrast to previous work [35], the new approach is sound. We implemented our approach as part of the open-source software-verification framework CPAchecker [12], and we perform an extensive experimental comparison of our implementation against the two existing tools that use k-induction and against other common software-verification approaches.

Contributions. We make the following contributions:

- a novel approach for providing *continuously-refined invariants* from data-flow analysis with precision adjustment in order to repeatedly inject invariants to k-induction,
- an *effective and efficient tool* implementation of a framework for software verification with k-induction that allows to express all existing approaches to k-induction in a *uniform, module-based, configurable architecture*, and
- an extensive *experimental evaluation* of (a) all approaches and their implementations in the framework, (b) the two existing k-induction tools Cbmc and Esbmc, and (c) the two different approaches predicate analysis and value analysis; the result being that the new technique outperforms all existing k-induction-based approaches to software verification.

Availability of Data and Tools. Our experiments are based on benchmark verification tasks from the 2015 Competition on Software Verification. All benchmarks, tools, and results of our evaluation are available on a supplementary web page[1].

[1] http://www.sosy-lab.org/~dbeyer/cpa-k-induction/
(successfully evaluated by the CAV 2015 Artifact Evaluation Committee)

```
1  int main() {
2    unsigned int x1 = 0, x2 = 0;
3    int s = 1;
4
5    while (nondet()) {
6      if (s == 1) x1++;
7      else if (s == 2) x2++;
8
9      s++;
10     if (s == 5) s = 1;
11
12     if ((s == 1) && (x1 != x2)) {
13       // Valid safety property
14       ERROR: return 1;
15     }
16   }
17 }
```

Fig. 1. Safe example program `example-safe`, which cannot be proven with existing k-induction-based approaches

```
1  int main() {
2    unsigned int x1 = 0, x2 = 0;
3    int s = 1;
4
5    while (nondet()) {
6      if (s == 1) x1++;
7      else if (s == 2) x2++;
8
9      s++;
10     if (s == 5) s = 1;
11   }
12
13   if (s >= 4) {
14     // Violation: s may be 4
15     ERROR: return 1;
16   }
17 }
```

Fig. 2. Unsafe example program `example-unsafe`, where some approaches may produce a wrong proof

Example. We illustrate the problem of k-induction that we address, and the strength of our approach, on two example programs. Both programs encode an automaton, which is typical, e.g., for software that implements a communication protocol. The automaton has a finite set of states, which is encoded by variable `s`, and two data variables `x1` and `x2`. There are some state-dependent calculations (lines 6 and 7 in both programs) that alternatingly increment `x1` and `x2`, and a calculation of the next state (lines 9 and 10 in both programs). The state variable cycles through the range from 1 to 4. These calculations are done in a loop with a non-deterministic number of iterations. Both programs also contain a safety property (the label `ERROR` should not be reachable). The program `example-safe` in Fig. 1 checks that in every fourth state, the values of `x1` and `x2` are equal; it satisfies the property. The program `example-unsafe` in Fig. 2 checks that when the loop exits, the value of state variable `s` is not greater or equal to 4; it violates the property.

First, note that the program `example-safe` is difficult or impossible to prove with many classical software-verification approaches other than k-induction: (1) BMC cannot prove safety for this program because the loop may run arbitrarily long. (2) Explicit-state model checking fails because of the huge state space (`x1` and `x2` can get arbitrarily large). (3) Predicate analysis with counterexample-guided abstraction refinement (CEGAR) and interpolation is able to prove safety, but only if the predicate $x1 = x2$ gets discovered. If the interpolants contain instead only predicates such as $x1 = 1$, $x2 = 1$, $x1 = 2$, etc., the predicate analysis will not terminate. Which predicates get discovered is hard to control and usually depends on internal interpolation heuristics of the satisfiability-modulo-theory (SMT) solver. (4) Traditional 1-induction is also not able to prove the program safe because the assertion is checked only in every fourth loop iteration (when `s` equals 1). Thus, the induction hypothesis is too weak (the program state `s = 4`, `x1 = 0`, `x2 = 1` is a counterexample for the step case in the induction proof).

Intuitively, this program should be provable by k-induction with a k of at least 4. However, for every k, there is a counterexample to the inductive-step case that refutes the proof. For such a counterexample, set `s = `$-k$, `x1 = 0`, `x2 = 1` at the beginning of the loop. Starting in this state, the program would increment `s` k times (induction hypothesis) and then reach `s = 1` with property-violating values of `x1` and `x2` in iteration $k+1$ (inductive step). It is clear that `s` can never be negative, but this fact is not present in the induction hypothesis, and thus, the proof fails. This illustrates the general problem of k-induction-based verification: safety properties often do not hold in unreachable parts of the state space of a program, and k-induction alone does not distinguish between reachable and unreachable parts of the state space. Therefore, approaches based on k-induction without auxiliary invariants will fail to prove safety for program `example-safe`.

This program could of course be verified more easily if it were rewritten to contain a stronger safety property such as $s \geq 1 \wedge s \leq 4 \wedge (s = 2 \Rightarrow x1 = x2+1) \wedge (s \neq 2 \Rightarrow x1 = x2)$ (which is a loop invariant and allows a proof by 1-induction without auxiliary invariants). However, our goal is to automatically verify real programs, and programmers usually neither write down trivial properties such as $s \geq 1$ nor more complex properties such as $s \neq 2 \Rightarrow x1 = x2$.

Our approach of combining k-induction with invariants proves the program safe with $k = 4$ and the invariant $s \geq 1$. This invariant is easy to find automatically using an inexpensive data-flow analysis, such as an interval analysis. For larger programs, a more complex invariant might be necessary, which might get generated at some point by our continuous strengthening of the invariant. Furthermore, stronger invariants can reduce the k that is necessary to prove a program. For example, the invariant $s \geq 1 \wedge s \leq 4 \wedge (s \neq 2 \Rightarrow x1 = x2)$ (which is still weaker than the full loop invariant above) allows to prove the program with $k = 2$. Thus, our strengthening of invariants can also shorten the inductive proof procedure and lead to better performance.

An existing approach tries to solve this problem of a too-weak induction hypothesis by initializing only the variables of the loop-termination condition to a non-deterministic value in the step case, and initializing all other variables to their initial value in the program [35]. However, this approach is not strong enough for the program `example-safe` and even produces a wrong proof (unsound result) for the program `example-unsafe`. This second example program contains a different safety property about `s`, which is violated. Because the variable `s` does not appear in the loop-termination condition, it is not set to an arbitrary value in the step case as it should be, and the inductive proof wrongly concludes that the program is safe because the induction hypothesis is too strong, leading to a missed bug and a wrong result. Our approach does not suffer from this unsoundness, because we add only invariants to the induction hypothesis that the invariant generation has proven to hold.

Related Work. The use of auxiliary invariants is a common technique in software verification [2,9,10,18,19,20,23,30,36], and techniques combining data-flow analysis and SMT solvers also exist [28,31]. In most cases, the purpose is to speed up the analysis. For k-induction, however, the use of invariants is crucial

in making the analysis terminate at all (cf. Fig. 1). There are several approaches to software verification using BMC in combination with k-induction.

Split-Case Induction. We use the *split-case k-induction* technique [26,27], where the base case and the step case are checked in separate steps. Earlier versions of Scratch [27] that use this technique transform programs with multiple loops into programs with only one single monolithic loop using a standard approach [1]. The alternative of recursively applying the technique to nested loops is discarded by the authors of Scratch [27], because the experiments suggested it was less efficient than checking the single loop that is obtained by the transformation. We also experimented with single-loop transformation, but our experimental results suggest that checking all loops at once in each case instead of checking the monolithic transformation result (which also encodes all loops in one) has no negative performance impact, so for simplicity, we omit the transformation. Scratch also supports *combined-case k-induction* [25], for which all loops are cut by replacing them with k copies each for the base and the step case, and setting all loop-modified variables to non-deterministic values before the step case. That way, both cases can be checked at once in the transformed program and no special handling for multiple loops is required. When using combined-case k-induction, Scratch requires loops to be manually annotated with the required k values, whereas its implementation of split-case k-induction supports iterative deepening of k as in our implementation. Contrary to Scratch, we do not focus on one specific problem domain [26,27], but want to provide a solution for solving a wide range of heterogeneous verification tasks.

Auxiliary Invariants. While both the split-case and the combined-case k-induction supposedly succeed with weaker auxiliary invariants than for example the inductive invariant approach [5], the approaches still do require auxiliary invariants in practice, and the tool Scratch requires these invariants to be annotated manually [25,27]. There are techniques for automatically generating invariants that may be used to help inductive approaches to succeed (e.g. [2,9,20]. These techniques, however, do not justify their additional effort because they are not guaranteed to provide the required invariants on time, especially if strong auxiliary invariants are required. Based on previous ideas of supporting k-induction with invariants generated by lightweight data-flow analysis [24], we therefore strive to leverage the power of the k-induction approach to succeed with auxiliary invariants generated by a data-flow analysis based on intervals. However, to handle cases where it is necessary to invest more effort into invariant generation, we increase the precision of these invariants over time.

Invariant Injection. A verification tool using a strategy similar to ours is PKind [28,33], a model checker for Lustre programs based on k-induction. In PKind, there is a parallel computation of auxiliary invariants, where candidate invariants derived by templates are iteratively checked via k-induction and, if successful, added to the set of known invariants [32]. While this allows for strengthening the induction hypothesis over time, the template-based approach lacks the flexibility that is available to an invariant generator using dynamic precision refinement [11], and the required additional induction proofs are

potentially expensive. We implemented checking candidate invariants with k-induction as a possible strategy of our invariant generation component.

Unsound Strengthening of Induction Hypothesis. ESBMC does not require additional invariants for k-induction, because it assigns non-deterministic values only to the loop-termination condition variables before the inductive-step case [35] and thus retains more information than our as well as the SCRATCH implementation [25,27], but k-induction in ESBMC is therefore potentially unsound. Our goal is to perform a real proof of safety by removing all pre-loop information in the step case, thus treating the unrolled iterations in the step case truly as "any k consecutive iterations", as is required for the mathematical induction. Our approach counters this lack of information by employing incrementally-refined invariant generation.

Parallel Induction. PKIND checks the base case and the step case in parallel, and ESBMC supports parallel execution of the base case, the forward condition, and the inductive-step case. In contrast, our base case and inductive-step case are checked sequentially, while our invariant generation runs in parallel to the base- and step-case checks.

2 *k*-Induction with Continuously-Refined Invariants

Our verification approach consists of two algorithms that run concurrently. One algorithm is responsible for generating program invariants, starting with an imprecise invariant, continuously refining (strengthening) the invariant. The other algorithm is responsible for finding error paths with BMC, and for constructing safety proofs with k-induction, for which it periodically picks up the new invariant that the former algorithm has constructed so far. The k-induction algorithm uses information from the invariant generation, but not vice versa. In our presentation, we assume that each program contains at most one loop; in our implementation, we handle programs with multiple loops by checking all loops together.

Iterative-Deepening *k*-Induction. Algorithm 1 shows our extension of the k-induction algorithm to a combination with continuously-refined invariants. Starting with an initial value for the bound k, e.g., 1, we iteratively increase the value of k after each unsuccessful attempt at finding a specification violation or proving correctness of the program using k-induction. The following description of our approach to k-induction is based on split-case k-induction [25], where for the propositional state variables s and s' within a state-transition system that represents the program, the predicate $I(s)$ denotes that s is an initial state, $T(s, s')$ states that a transition from s to s' exists, and $P(s)$ asserts the safety property for the state s.

Base Case. Lines 3 to 5 implement the *base case*, which consists of running BMC with the current bound k. This means that starting from an initial program state, all paths of the program up to a maximum path length $k-1$ are explored. If an error path is found, the algorithm terminates.

Algorithm 1 Iterative-Deepening k-Induction

Input:
the initial value $k_{init} \geq 1$ for the bound k,
an upper limit k_{max} for the bound k,
a function $\mathsf{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \mathsf{inc}(n) > n$ for increasing the bound k,
the initial states defined by the predicate I,
the transfer relation defined by the predicate T, and
a safety property P
Output: **true** if P holds, **false** otherwise

1: $k := k_{init}$
2: **while** $k \leq k_{max}$ **do**
3: $\quad base\_case := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left(\bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg P(s_n) \right)$
4: $\quad$ **if** $\mathsf{sat}(base\_case)$ **then**
5: $\quad\quad$ **return false**
6: $\quad forward\_condition := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$
7: $\quad$ **if** $\neg\mathsf{sat}(forward\_condition)$ **then**
8: $\quad\quad$ **return true**
9: $\quad step\_case_n := \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$
10: $\quad$ **repeat**
11: $\quad\quad Inv := \mathsf{get\_currently\_known\_invariant}()$
12: $\quad\quad$ **if** $\neg\mathsf{sat}(Inv(s_n) \wedge step\_case_n)$ **then**
13: $\quad\quad\quad$ **return true**
14: $\quad$ **until** $Inv = \mathsf{get\_currently\_known\_invariant}()$
15: $\quad k := \mathsf{inc}(k)$
16: **return unknown**

Algorithm 2 Continuous Invariant Generation using Configurable Program Analysis

Input:
a configurable program analysis with dynamic precision adjustment $\mathbb{D}$,
the initial states defined by predicate I,
a coarse initial precision π_0,
a safety property P
Output: **true** if P holds

1: $\pi := \pi_0$
2: $Inv := true$
3: **loop**
4: $\quad reached := \mathsf{CPAAlgorithm}(\mathbb{D}, I, \pi)$
5: $\quad$ **if** $\forall s \in reached : P(s)$ **then**
6: $\quad\quad$ **return true**
7: $\quad Inv := Inv \wedge \bigvee_{s \in reached} s$
8: $\quad \pi := \mathsf{RefinePrec}(\pi, reached)$

Forward Condition. Otherwise we check whether there exists a path with length $k' > k - 1$ in the program, or whether we have already fully explored the state space of the program (lines 6 to 8). In the latter case the program is safe and the algorithm terminates. This check is called the *forward condition* [29].

Inductive Step. Checking the forward condition can, however, only prove safety for programs with finite (and short) loops. Therefore, the algorithm also attempts an inductive proof (lines 9 to 14). The *inductive-step case* checks if, after every sequence of k loop iterations without a property violation, there is also no property violation before loop iteration $k+1$. For model checking of software, however, this check would often fail inconclusively without auxiliary invariants [8]. In our approach, we make use of the fact that the invariants that were generated so far by the concurrently-running invariant-generation algorithm hold, and conjunct these facts to the induction hypothesis. Thus, the inductive-step case proves a program safe if the following condition is unsatisfiable:

$$Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$$

where Inv is the currently available program invariant, and $s_n, \ldots, s_{n+k}$ is any sequence of states. If this condition is satisfiable, then the induction check is inconclusive, and the program is not yet proved safe or unsafe with the current value of k and the current invariant. If during the time of the satisfiability check of the step case, a new (stronger) invariant has become available (condition in line 14 is false), we immediately re-check the step case with the new invariant. This can be done efficiently using an incremental SMT solver for the repeated satisfiability checks in line 12. Otherwise, we start over with an increased value of k.

Note that the inductive-step case is similar to a BMC check for the presence of error paths of length exactly $k + 1$. However, as the step case needs to consider any consecutive $k + 1$ loop iterations, and not only the first such iterations, it does not assume that the execution of the loop iterations begins in an initial state. Instead, it assumes that there is a sequence of k iterations without any property violation (induction hypothesis).

Continuous Invariant Generation. Our continuous invariant generation incrementally produces stronger and stronger program invariants. It is based on iterative refinement, each time using an increased precision. After each strengthening of the invariant, it can be used as injection invariant by the k-induction procedure. It may happen that this analysis proves safety of the program all by itself, but this is not its main purpose here.

Our k-induction module works with any kind of invariant-generation procedure, as long as its precision, i.e., its level of abstraction, is configurable. We implemented two different invariant-generation approaches: KI and DF, described below.

We use the design of Fig. 3 to explain our flexible and modular framework for k-induction: k-induction is a verification technique, i.e., an invariant generation. In this paper, the main algorithm is thus the k-induction, as defined in Algorithm 1. We denote the algorithm by KI. If invariants are generated and injected into KI, we denote this injection by KI←. Thus, the use of generated invariants that are produced by a data-flow analysis (DF) are denoted by KI←DF. If the invariant generator continuously refines the invariants and repeatedly *injects* those invariants into KI, this is denoted by KI←↻, more specifically, if data-flow analysis with dynamic precision adjustment (our new contribution) is used, we have KI←↻DF, and if the PKind approach is used, i.e., KI is used to construct invariants, we have KI←↻KI. Now, since the second KI, which constructs invariants for injection into the first KI, can again get invariants injected, we can further build an approach KI←↻KI←↻DF that combines all approaches such that the invariant-generating KI benefits from the invariants generated with DF, and the main KI algorithm that tries to prove program safety benefits from both invariant generators.

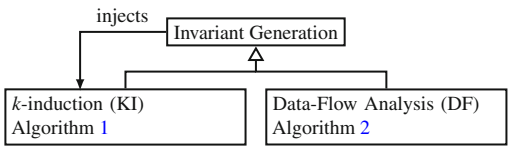


Fig. 3. Configurable design of a k-induction framework

KI. PKind [33] introduced the idea to construct invariants for injection in parallel, using a template-based method that extracts candidate invariants from the program and verifies their validity using k-induction [32]. If the candidate invariants are found to be valid, they are injected to the main k-induction procedure. We re-implemented the PKind approach in our framework (KI←↻KI), using a separate instance of k-induction to prove candidate invariants. Being based on k-induction, the power of this technique is continuously increased by increasing k. We derive the candidate invariants by taking the negations of assumptions on the control-flow paths to error locations. Similar to our Algorithm 2, each time this k-induction algorithm succeeds in proving a candidate invariant, the previously-known invariant is strengthened with this newly generated invariant. In our tool, we used an instance of Algorithm 1 to implement this approach. We are thus able to further combine this technique with other auxiliary invariant-generation approaches.

DF. As a second invariant-generation approach (our contribution), we use the reachability algorithm CPAAlgorithm for configurable program analysis with dynamic precision adjustment [11]. Algorithm 2 shows our continuous invariant generation. The initial program invariant is represented by the formula *true*. We start with running the invariant-generating analysis once with a coarse initial precision (line 4). After each run of the program-invariant generation, we strengthen the previously-known program invariant with the newly-generated invariant (line 7, note that the program invariant *Inv* is not a safety invariant) and announce it globally (such that the k-induction algorithm can inject it).

If the analysis was able to prove safety of the program, the algorithm terminates (lines 5 to 6). Otherwise, the analysis is restarted with a higher precision. The CPAAlgorithm takes as input a configurable program analysis (CPA), a set of initial abstract states, and a precision. It returns a set of reachable abstract states that form an over-approximation of the reachable program states. Depending on the used CPA and the precision, the analysis by CPAAlgorithm can be efficient and abstract like data-flow analysis or expensive and precise like model checking.

For invariant generation, we choose an abstract domain based on expressions over intervals [8]. Note that this is not a requirement of our approach, which works with any kind of domain. Our choice is based on the high flexibility of this domain, which can be fast and efficient as well as precise. For this CPA, the precision is a triple (Y, n, w), where $Y \subseteq X$ is a specific selection of important program variables, n is the maximal nesting depth of expressions in the abstract state, and w is a boolean specifying whether widening should be used. Those variables that are considered important will not be over-approximated by joining abstract states. With a higher nesting depth, more precise relations between variables can be represented. The use of widening ensures timely termination (at the expense of a lower precision), even for programs with loops with many iterations, like those in the examples of Figs. 1 and 2. An in-depth description of this abstract domain is presented in a technical report [8].

3 Experimental Evaluation

We implemented all existing approaches to k-induction, compare all configurations with each other, and the best configuration with other k-induction-based software verifiers, as well as to two standard approaches to software verification: predicate and value analysis.

Benchmark Verification Tasks. As benchmark set we use verification tasks from the 2015 Competition on Software Verification (SV-COMP'15) [7]. We took all 3 964 verification tasks from the categories *ControlFlow*, *DeviceDrivers64*, *HeapManipulation*, *Sequentialized*, and *Simple*. The remaining categories were excluded because they use features (such as bit-vectors, concurrency, and recursion) that not all configurations of our evaluation support. A total of 1 148 verification tasks in the benchmark set contain a known specification violation. Although we cannot expect an improvement for these verification tasks when using auxiliary invariants, we did not exclude them because this would unfairly give advantage to the new approach (which spends some effort generating invariants, which are not helpful when proving existence of an error path).

Experimental Setup. All experiments were conducted on computers with two 2.6 GHz 8-Core CPUs (Intel Xeon E5-2560 v2) with 135 GB of RAM. The operating system was Ubuntu 14.04 (64 bit), using Linux 3.13 and OpenJDK 1.7. Each verification task was limited to two CPU cores, a CPU run time of 15 min, and

a memory usage of 15 GB. The benchmarking framework BenchExec[2] ensures precise and reproducible results.

Presentation. All benchmarks, tools, and the full results of our evaluation are available on a supplementary web page.[3] All reported times are rounded to two significant digits. We use the scoring scheme of SV-COMP'15 to calculate a score for each configuration. For every real bug found, 1 point is assigned, for every correct safety proof, 2 points are assigned. A score of 6 points is subtracted for every wrong alarm (false positive) reported by the tool, and 12 points are subtracted for every wrong proof of safety (false negative). This scoring scheme values proving safety higher than finding error paths, and significantly punishes wrong answers, which is in line with the community consensus [7] on difficulty of verification vs. falsification and importance of correct results. We consider this a good fit for evaluating an approach such as k-induction, which targets at producing safety proofs.

In Figs. 4 and 5, we present experimental results using a plot of quantile functions for accumulated scores as introduced by the Competition on Software Verification [6], which shows the score and CPU time for successful results and the score for wrong answers. A data point (x, y) of a graph means that for the respective configuration the sum of the scores of all wrong answers and the scores for all correct answers with a run time of less than or equal to y seconds is x. For the left-most point (x, y) of each graph, the x-value shows the sum of all negative scores for the respective configuration and the y-value shows the time for the fastest successful result. For the right-most point (x, y) of each graph, the x-value shows the total score for this configuration, and the y-value shows the maximal run time. A configuration can be considered better, the further to the right (the closer to 0) its graph begins (fewer wrong answers), the further to the right it ends (more correct answers), and the lower its graph is (less run time).

Comparison of *k*-Induction-Based Approaches. We implemented all approaches in the Java-based open-source software-verification framework CPAchecker [12], which is available online[4] under the Apache 2.0 License. For the experiments, we used version `1.4.5-cav15` of CPAchecker, with SMTInterpol [21] as SMT solver (using uninterpreted functions and linear arithmetic over integers and reals). The k-induction algorithm of CPAchecker was configured to increment k by 1 after each try (in Algorithm 1, $\mathsf{inc}(k) = k+1$). The precision refinement of the DF-based continuous invariant generation (Algorithm 2) was configured to increment the number of important program variables in the first, third, fifth, and any further precision refinements. The second precision refinement increments the expression-nesting depth, and the fourth disables the widening.

[2] https://github.com/dbeyer/benchexec
[3] http://www.sosy-lab.org/~dbeyer/cpa-k-induction/
[4] http://cpachecker.sosy-lab.org

Table 1. Results of k-induction-based configurations in CPAchecker for all 3 964 verification tasks with different approaches for generating auxiliary invariants

Approach	KI	KI←DF $(0,1,t)$	KI←DF $(8,2,t)$	KI←DF $(16,2,t)$	KI←DF $(16,2,f)$	KI↞↻KI	KI↞↻DF	KI↞↻KI↞↻DF
Score	2 246	3 944	4 117	4 062	3 992	3 535	4 249	**4 282**
Correct results	1 531	2 377	2 462	2 428	2 392	2 169	2 507	**2 519**
Wrong proofs	1	1	2	1	1	1	1	1
Wrong alarms	30	30	30	30	30	30	26	25
CPU time (h)	530	330	330	340	340	380	320	320
Wall time (h)	440	240	210	210	210	270	190	170
Times for correct results only:								
CPU time (h)	17	32	39	36	36	28	36	41
Wall time (h)	13	19	22	20	20	18	20	22
k-Values for correct safe results only:								
Max. final k	101	101	100	100	126	101	112	111
Avg. final k	1.7	1.4	1.7	1.8	1.8	1.8	1.8	1.9

We evaluated the following groups of k-induction approaches: (1) without any auxiliary invariants (KI), (2) with auxiliary invariants of different precisions generated by the DF approach (KI←DF), and (3) with continuously-refined invariants (KI↞↻).

The k-induction-based configuration using no auxiliary invariants (KI) is an instance of Algorithm 1 where get_currently_known_invariant() always returns *true* as invariant and Algorithm 2 does not run at all.

The configurations using generated invariants (KI←DF) are also instances of Algorithm 1. Here, Algorithm 2 runs in parallel, however, it terminates after one loop iteration. We denote these configurations with triples (s,n,w) that represent the precision (Y,n,w) of the invariant generation with s being the size of the set of important program variables ($s = |Y|$). For example, the first of these configurations, $(0,1,\mathit{true})$, has no variables in the set Y of important program variables (i.e., all variables get over-approximated by the merge operator), the maximum nesting depth of expressions in the abstract state is 1, and the widening operator is used. The remaining configurations we use are $(8,2,\mathit{true})$, $(16,2,\mathit{true})$, and $(16,2,\mathit{false})$. These configurations were selected because they represent some of the extremes of the precisions that are used during dynamic invariant generation. It is impossible to cover every possible valid configuration within the scope of this paper.

There are three configurations using continuously-refined invariants: (1) using the k-induction approach similar to PKind to generate invariants, refining by increasing k, denoted as KI↞↻KI, (2) using the DF-based approach to generate invariants, refining by precision adjustment, denoted as KI↞↻DF, and (3) using both approaches in parallel combination, denoted as KI↞↻KI↞↻DF. All configurations using invariant generation run the generation in parallel to the main k-induction algorithm, an instance of Algorithm 1.

Score and Reported Results. The configuration KI with no invariant generation receives the lowest score of 2 246, and (as expected) can verify only 1 531 programs successfully. This shows that it is indeed important in practice to enhance k-induction-based software verification with invariants. The configurations KI←DF using invariant generation produce similar numbers of correct results (around 2 400), improving upon the results of the plain k-induction without auxiliary invariants by a score of 1 700 to 1 800. Even though these configurations solve a similar number of programs, a closer inspection reveals that each of the configurations is able to correctly solve significant amounts of programs where the other configurations run into timeouts. This observation explains the high score of 4 249 points achieved by our approach of injecting the continuously-refined invariants generated with data-flow analysis into the k-induction engine (configuration KI←↺-DF). By combining the advantages of fast and coarse precisions with those of slow but fine precisions, it correctly solves 2 507 verification tasks, which is 45 more than the best of the chosen configurations without dynamic refinement. Using a k-induction-based invariant generation as done by PKind (configuration KI←↺-KI) is also a successful technique for improving the amount of solvable verification tasks, and thus, combining both invariant-generation approaches with continuously refining their precision and injecting the generated invariants into the k-induction engine (configuration KI←↺-KI←↺-DF) is the most effective of all evaluated k-induction-based approaches, with a score of 4 282, and 2 519 correct results. The few wrong proofs produced by the configurations are not due to conceptual problems, but only due to incompleteness in the analyzer's handling of certain constructs such as unbounded arrays and pointer aliasing.

Performance. Table 1 shows that by far the largest amount of time is spent by the configuration KI (no auxiliary invariants), because for those programs that cannot be proved without auxiliary invariants, the k-induction procedure loops incrementing k until the time limit is reached. The wall times and CPU times for the correct results correlate roughly with the amount of correct results, i.e., on average about the same amount of time is spent on correct verifications, whether or not invariant generation is used. This shows that the overhead of generating auxiliary invariants is well-compensated.

The configurations with invariant generation have a relatively higher CPU time compared to their wall time because these configurations spend some time generating invariants in parallel to the k-induction algorithm. The results show, however, that the time spent for the continuously-refined invariant generation clearly pays off as the configuration using both data-flow analysis and k-induction for invariant generation is not only the one with the most correct results, but at the same time one of the two fastest configurations with only 320 h in total. Even though they produced much more correct results, the configurations KI←↺-KI←↺-DF and KI←↺-DF did not exceed the times of the chosen configurations using invariant generation without continuous refinement. The configuration KI←↺-KI using only k-induction to continuously generate invariants is slower, but produces results for some programs where the configuration

Table 2. Results of k-induction-based tools for all 3 964 verification tasks

Tool	CBMC	ESBMC		CPACHECKER
Configuration		sequential	parallel	KI⇜↺⇝KI⇜↺⇝DF
Score	−4 372	1 674	1 716	**4 282**
Correct results	1 949	2 050	2 059	**2 519**
Wrong proofs	666	156	152	**1**
Wrong alarms	**5**	9	13	25
CPU time (h)	360	290	370	320
Wall time (h)	360	290	200	170
Times for correct results only:				
CPU time (h)	3.9	16	26	41
Wall time (h)	3.9	16	13	22
k-Values for correct safe results only:				
Max. final k	50	2 048	1 952	111
Avg. final k	1.1	5.3	7.1	1.9

KI⇜↺⇝DF fails. The results show that the combination of the techniques reaps the benefits of both.

These results show that the additional effort invested in generating auxiliary invariants is well-spent, as it even decreases the overall time due to the fewer timeouts. As expected, the continuously-refined invariants solve many tasks quicker than the configurations using invariant generation with high precisions and without refinement.

Final value of k. The bottom of Table 1 shows some statistics about the final values of k for the correct safety proofs. There are only small differences between the maximum k values of most of the configurations. Interestingly, the configuration using non-dynamic invariant generation with high precision has a higher maximum final value of k than the others, because for the verification task `afnp2014_true-unreach-call.c.i`, a strong invariant generated only with this configuration allowed the proof to succeed. This effect is also observable in the continuously-refined configurations using invariants generated by data-flow analysis: They are also able to solve this verification task, and, by dynamically increasing the precision, find the required auxiliary invariant even earlier with loop bounds 112 and 111, respectively. There is also a verification task in the benchmark set, `gj2007_true-unreach-call.c.i`, where most configurations need to unroll a loop with bound 100 to prove safety, while the strong invariant generation technique allows the proof to succeed earlier, at a loop bound of 16. The continuously-refined configurations benefit from the same effect: KI⇜↺⇝DF and KI⇜↺⇝KI⇜↺⇝DF solve this task at loop bounds 22 and 19, respectively.

Comparison with Other Tools. For comparison with other k-induction-based tools, we evaluated ESBMC and CBMC, two software model checkers with support for k-induction. For CBMC, we used version 5.1 in combination with a wrapper script for split-case k-induction provided by M. Tautschnig. For ESBMC we used version 1.25.2 in combination with a wrapper script that enables k-induction (based on the SV-COMP'13 submission [35]). We also provide results for the experimental parallel k-induction of ESBMC, but note that our benchmark setup is not focused on parallelization (using only two CPU cores and a CPU-time limit instead of wall time). The CPACHECKER configuration in this comparison is the one with continuously-refined invariants and both invariant generators (KI⟲-KI⟲-DF). Table 2 gives the results; Fig. 4 shows the quantile functions of the accumulated scores for each configuration. The results for CBMC are not competitive, which may be attributed to the experimental nature of its k-induction support.

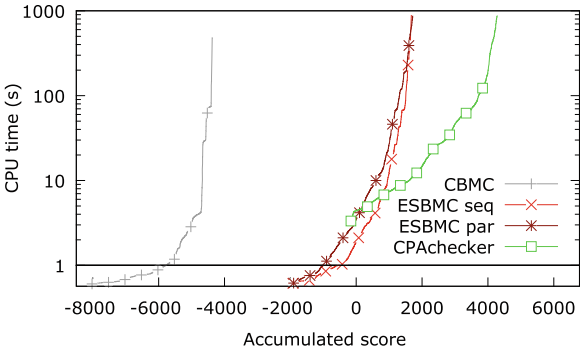


Fig. 4. Quantile functions of k-induction-based tools (CPACHECKER in configuration KI⟲-KI⟲-DF) for accumulated scores showing the CPU time for the successful results; linear scale between 0 s and 1 s, logarithmic scale beyond

Score. CPACHECKER in configuration KI⟲-KI⟲-DF successfully verifies almost 500 tasks (20 %) more than ESBMC. Furthermore, it has only 1 missed bug, which is related to unsoundness in the handling of some C features, whereas ESBMC has more than 150 wrong safety proofs. This large number of wrong results must be attributed to the unsound heuristic of ESBMC for strengthening the induction hypothesis, where it retains potentially incorrect information about loop-modified variables [35]. We have previously also implemented this approach in CPACHECKER and obtained similar results [8]. The large number of wrong proofs reduces the confidence in the soundness of the correct proofs. Consequently, the score achieved by CPACHECKER in configuration KI⟲-KI⟲-DF is much higher than the score of ESBMC (4 282 compared to 1 674 points). This clear advantage is also visible in Fig. 4. The parallel version of ESBMC performs somewhat better than its sequential version, and misses fewer bugs. This is due to the fact that the base case and the step case are performed in parallel, and the loop bound k is incremented independently for each of them. The base case is usually easier to solve for the SMT solver, and thus the base-case checks proceed faster than the step-case checks (reaching a higher value of k sooner). Therefore, the parallel version manages to find some bugs by reaching the relevant k in the base-case checks earlier than in the step-case checks, which would produce

a wrong safety proof at reaching k. However, the number of wrong proofs is still much higher than with our approach, which is conceptually sound. Thus, the score of the new, sound approach is more than 2 500 points higher.

Performance. Table 2 shows that our approach needs only 10 % more CPU time than the sequential version of ESBMC for solving a much higher number of tasks, and even needs less CPU and wall time than the parallel version of ESBMC. This indicates that due to our invariants, we succeed more often with fewer loop unrollings, and thus in less time. It also shows that the effort invested for generating the invariants is well spent.

Final Value of k. The bottom of Table 2 contains some statistics on the final value of k that was needed to verify a program. The table shows that for safe programs, CPACHECKER needs a loop bound that is (on average) only about one third of the loop bound that ESBMC needs. This advantage is due to the use of generated invariants, which make the induction proofs easier and likely to succeed with a smaller number of k. The verification task `array_true-unreach-call2.i` is solved by ESBMC after completely unwinding the loop, therefore reaching the large k-value 2 048. In the parallel version, the (quicker) detached base case hits this bound while the inductive step case is still at $k = 1\,952$.

Comparison with Other Approaches. We also compare our combination of k-induction with continuously-refined invariants with other common approaches for software verification. We use for comparison two analyses based on CEGAR, a predicate analysis [13] and a value analysis [14]. Both are implemented in CPACHECKER, which allows us to compare the approaches inside the same tool, using the same run-time environment, SMT solver, etc., and focus only on the conceptual differences between the analyses.

Figure 5 shows a quantile plot to compare the configuration KI⟲-KI⟲-DF with CPACHECKER predicate analysis and value analysis. The predicate analysis solves 2 463 verification tasks in a total of 280 CPU hours, and achieves a score of 4 201. The value analysis solves 2 367 verification tasks in a total of 303 CPU hours, and achieves a score of 4 216 because it has a few wrong results less. The higher number of solved tasks (2 519) and the higher score (4 282)

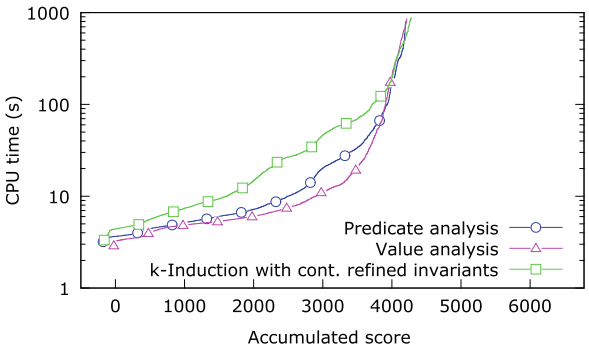


Fig. 5. Quantile functions of different approaches implemented in CPACHECKER (k-induction in configuration KI⟲-KI⟲-DF) for accumulated scores showing the CPU time for the successful results

of the k-induction-based configuration show that k-induction is clearly competitive with the state-of-the-art in software verification, if it is boosted by injecting continuously-refined invariants.

4 Conclusion

We have presented the novel idea of injecting invariants into k-induction that are generated using data-flow analysis with dynamic precision adjustment, and contribute a publicly available implementation of our idea within the software-verification framework CPAchecker. Our extensive experiments show that the new approach outperforms all existing implementations of k-induction for software verification, and that it is competitive compared to other, more mature techniques for software verification. We showed that a sound, effective, and efficient k-induction approach to general-purpose software verification is possible, and that the additional resources required to achieve these combined benefits are negligible if invested judiciously. At the same time, there is still room for improvement of our technique. An interesting improvement would be to add an information flow between the two cooperating algorithms in the reverse direction. If the k-induction procedure could tell the invariant generation which facts it misses to prove safety, this could lead to a more efficient and effective approach to generate invariants that are specifically tailored to the needs of the k-induction proof. Already now, CPAchecker is parsimonious in terms of unrollings, compared to other tools. The low k-values required to prove many programs show that even our current invariant generation is powerful enough to produce invariants that are strong enough to help cut down the necessary number of loop unrollings. k-induction-guided precision refinement might direct the invariant generation towards providing weaker but still useful invariants for k-induction more efficiently.

Acknowledgments. We thank M. Tautschnig and L. Cordeiro for explaining the optimal available parameters for k-induction, for the verifiers Cbmc and Esbmc, respectively.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Awedh, M., Somenzi, F.: Automatic invariant strengthening to prove properties in bounded model checking. In: Proceedings of DAC, pp. 1073–1076. ACM/IEEE (2006)
3. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: technology transfer of formal methods inside microsoft. In: Proceedings of IFM, LNCS, vol. 2999, pp. 1–20. Springer (2004)
4. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011)
5. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of PASTE, pp. 82–87. ACM (2005)

6. Beyer, D.: Second competition on software verification. In: Proceedings of TACAS, LNCS, vol. 7795, pp. 594–609. Springer (2013)
7. Beyer, D.: Software verification and verifiable witnesses. In: Proceedings of TACAS, LNCS, vol. 9035, pp. 401–416. Springer (2015)
8. Beyer, D., Dangl, M., Wendler, P.: Combining k-induction with continuously-refined invariants. Technical Report MIP-1503, University of Passau, January 2015. arXiv:1502.00096
9. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Proceedings of VMCAI, LNCS, vol. 4349, pp. 378–394. Springer (2007)
10. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Procedings of PLDI, pp. 300–309. ACM (2007)
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proceedings of ASE, pp. 29–38. IEEE (2008)
12. Beyer, D., Keremoglu, M.:CPAchecker: A tool for configurable software verification. In: Proceedings of CAV, LNCS, vol. 6806, pp. 184–190. Springer (2011)
13. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proceedings of FMCAD, pp. 189–197. FMCAD (2010)
14. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proceedings of FASE, LNCS, vol. 7793, pp. 146–162. Springer (2013)
15. Biere, A.: Handbook of Satisfiability. IOS Press, Amsterdam (2009)
16. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
17. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of TACAS, LNCS, vol. 1579, pp. 193–207. Springer (1999)
18. Bjørner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theor. Comput. Sci. **173**(1), 49–87 (1997)
19. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of PLDI, pp. 196–207. ACM (2003)
20. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. FAC **20**(4–5), 379–405 (2008)
21. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Proceedings of SPIN, LNCS, vol. 7385, pp. 248–254. Springer (2012)
22. Cordeiro, L., Fischer, B., Silva, J.P.M.: SMT-based bounded model checking for embedded ANSI-C software. In: Proceedings of ASE, pp. 137–148. IEEE (2009)
23. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Procedings of POPL, pp. 84–96 (1978)
24. Donaldson, A.F., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In: Proceedings of VMCAI, LNCS, vol. 6538, pp. 169–183. Springer, Heidelberg (2011)
25. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Proceeding of Static Analysis. LNCS, vol. 6887, pp. 351–368. Springer (2011)
26. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Proceedings of TACAS, LNCS, vol. 6015, pp. 280–295. Springer (2010)
27. Donaldson, A.F., Kröning, D., Rümmer, P.: Automatic analysis of DMA races using model checking and k-induction. FMSD **39**(1), 83–113 (2011)

28. Garoche, P.-L., Kahsai, T., Tinelli, C.: Incremental invariant generation using logic-based automatic abstract transformers. In: Proceedings of NFM, LNCS, vol. 7871, pp. 139–154. Springer (2013)
29. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: Proceedings of MEMOCODE, pp. 113–122. IEEE (2010)
30. Gupta, A., Rybalchenko, A.: InvGen: an efficient invariant generator. In: Proceedings of CAV, LNCS, vol. 5643, pp. 634–640. Springer (2009)
31. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: UFO: verification with interpolants and abstract interpretation. In: Proceedings of TACAS, LNCS, vol. 7795, pp. 637–640. Springer (2013)
32. Kahsai, T., Ge, Y., Tinelli, C.: Instantiation-based invariant discovery. In: Proceedings of NFM, LNCS, vol. 6617, pp. 192–206. Springer (2011)
33. Kahsai, T., Tinelli, C.: Pkind: a parallel k-induction based model checker. In: Proceedings of International Workshop on Parallel and Distributed Methods in Verification, EPTCS 72, pp. 55–62 (2011)
34. Khoroshilov, A., Mutilin, V., Petrenko, A., Zakharov, V.: Establishing linux driver verification process. In: Proceedings of PSI, LNCS, vol. 5947, pp. 165–176. Springer (2010)
35. Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Handling unbounded loops with ESBMC 1.20. In: Proceedings of TACAS, LNCS, vol. 7795, pp. 619–622. Springer (2013)
36. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Proceedings of VMCAI, LNCS, vol. 3385, pp. 25–41. Springer (2005)
37. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proceedings of FMCAD, LNCS, vol. 1954, pp. 108–125. Springer (2000)
38. Wahl, T.: The k-induction principle (2013). http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf

J Autom Reasoning (2018) 60:299–335
https://doi.org/10.1007/s10817-017-9432-6

A Unifying View on SMT-Based Software Verification

Dirk Beyer[1] · **Matthias Dangl**[1] · **Philipp Wendler**[1]

Received: 1 September 2017 / Accepted: 5 September 2017 / Published online: 4 December 2017

Abstract After many years of successful development of new approaches for software verification, there is a need to consolidate the knowledge about the different abstract domains and algorithms. The goal of this paper is to provide a compact and accessible presentation of four SMT-based verification approaches in order to study them in theory and in practice. We present and compare the following different "schools of thought" of software verification: bounded model checking, k-induction, predicate abstraction, and lazy abstraction with interpolants. Those approaches are well-known and successful in software verification and have in common that they are based on SMT solving as the back-end technology. We reformulate all four approaches in the unifying theoretical framework of configurable program analysis and implement them in the verification framework CPACHECKER. Based on this, we can present an evaluation that thoroughly compares the different approaches, where the core differences are expressed in configuration parameters and all other variables are kept constant (such as parser front end, SMT solver, used theory in SMT formulas). We evaluate the effectiveness and the efficiency of the approaches on a large set of verification tasks and discuss the conclusions.

Keywords Software verification · Program analysis · Bounded model checking · k-induction · IMPACT · Lazy abstraction · Predicate abstraction · SMT solving

1 Introduction

In recent years, advances in automatic methods for software verification have lead to an increased effort towards applying software verification to industrial systems, in particular operating-systems code [5,8,24,56]. Predicate abstraction [47] with counterexample-guided abstraction refinement (CEGAR) [34] and lazy abstraction [51], lazy abstraction with inter-

A preliminary version of this article was published in Proceedings of VSTTE 2016 [12].

[1] LMU Munich, Munich, Germany

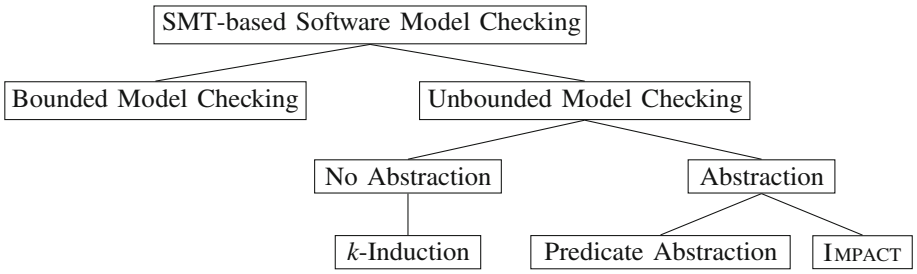


Fig. 1 Classification of approaches

polants [61], large-block encoding [11,21], and k-induction with auxiliary invariants [13,41] are some of the concepts that helped scale verification technology from simple example programs to real-world software. In the 6th International Competition on Software Verification (SV-COMP'17) [10], nine out of the 15 candidates participating in category *Overall* used some of these techniques, and out of the remaining six, four are bounded model checkers [26]. Considering this apparent success, we revisit an earlier work that presented a unifying algorithm for lazy predicate abstraction (BLAST-like) and lazy abstraction with interpolants (IMPACT-like) and showed that both techniques perform similarly [25]. We extend this unifying framework to bounded model checking and k-induction and conduct a comparative evaluation of bounded model checking, k-induction, lazy predicate abstraction, and lazy abstraction with interpolants. We observe that the previously drawn conclusions about the two lazy-abstraction techniques still hold today and show that even though abstraction is often necessary for scalability, k-induction has the potential to outperform the other two techniques. We restrict our presentation to safety properties; however, the techniques that we present can be used also for checking liveness [67].

Unfortunately, there is not much work available on rigorous comparison of algorithms. General overviews over methods for reasoning [9] and of approaches for software model checking [53] exist, but no systematic comparison of the algorithms in a common setting. This paper formulates four widely used SMT-based approaches for software verification in a common theoretical framework and tool implementation and compares their effectiveness and efficiency. Figure 1 tries to classify the approaches; in the following we use this structure also to give pointers to other implementations of the approaches.

1.1 Bounded Model Checking

Many software bugs can be found by a bounded search through the state space of the program. Bounded model checking [26] for software encodes all program paths that result from a bounded unrolling of the program in an SMT formula that is satisfiable if the formula encodes a feasible program path from the program entry to a violation of the specification. Several implementations were demonstrated to be successful in improving software quality by revealing program bugs (especially on short paths), for example CBMC [35], ESBMC [37], LLBMC [69], and SMACK [64]. The characteristics to quickly verify even a large portion of the state space of many types of programs without the need of computing expensive abstractions made the technique a basis component in many verification tools (cf. Table 4 in the report for SV-COMP'17 [10]).

1.2 Unbounded without Abstraction[1]

The idea of bounded model checking (to encode portions of a program as SMT formula, even if they are large) can be used also for unbounded verification by using an induction argument [68], i.e., checking whether the safety property is implied by all paths from the program entry to the loop head and after assuming the safety property at the loop head (induction hypothesis) by all paths through the loop body. Because the safety property is often not inductive, the more general k-induction principle [70] is used. The approach of k-induction is implemented in CBMC [35], CPACHECKER [13], ESBMC [65], PKIND [55], and 2LS [66]. The approach of strengthening k-induction proofs with continuously refining invariant generation [13] was independently reproduced later in 2LS [29].

1.3 Unbounded with Abstraction

A completely different approach is to compute an overapproximation of the state space, using insights from data-flow analysis [1,57,63]. While overapproximation can be a useful technique for mitigating the problem of state-space explosion, a too coarse level of abstraction may cause false alarms. Therefore, state-space abstraction is often combined with counterexample-guided abstraction refinement (CEGAR) [34] and lazy abstraction refinement [51]. Several verifiers implement a predicate abstraction [47]: for example, SLAM [6], BLAST [15], and CPACHECKER [20]. A safe inductive invariant is computed by iteratively refining the abstract states, where new predicates are discovered during each CEGAR step. Interpolation [38,60] is a successful method to obtain useful predicates from infeasible error paths; path invariants [17] can be used to obtain loop invariants for path programs.

Instead of using predicate abstraction, it is possible to construct the abstract state space directly from interpolants using the IMPACT algorithm [61].

1.4 Structure

In the remainder of this article, we first describe some necessary background in Sect. 2 and define a configurable program analysis as the foundation for unifying SMT-based approaches for software verification in Sect. 3. In Sect. 4, we express the four approaches within our framework and explain their core concepts and respective differences. Section 5 contains an experimental study of the effectiveness and efficiency of the presented approaches on a large set of verification tasks.

2 Background

2.1 Program Representation

In this section we provide basic definitions from the literature [15]. For simplicity, we restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[2] Such a program can be represented using a *control-flow automaton* (CFA), which is a directed graph with

[1] Strictly speaking, every verification technique attempts to construct an abstraction in the sense that a successful safety proof would establish the safety property as a valid abstraction of the program. In this classification, we differentiate between abstraction techniques that deliberately construct an abstract model of the program from derived abstract facts and non-abstraction techniques that aim to prove the safety property without constructing such an (auxiliary) abstract model of any kind.

[2] Our implementation is based on CPACHECKER [20], which supports C programs.

program operations attached to its edges. A CFA $A = (L, l_{INIT}, G)$ consists of a set L of *program locations*, an initial location $l_{INIT} \in L$ that represents the program entry point, and a set $G \subseteq (L \times Ops \times L)$ of edges between program locations, each labeled with an operation that is executed when the control flows along the edge. The set of all program variables that occur in the operations of a CFA is denoted by X. A *concrete data state* $c : X \to \mathbb{Z}$ is a mapping from program variables to integers. A set of concrete data states is called *region*. We represent regions using first-order formulas ψ over variables from X such that the set $[\![\psi]\!]$ of concrete data states that is represented by ψ is defined as $\{c \mid c \models \psi\}$. A *concrete state* $(c, l) : (X \to \mathbb{Z}) \times L$ is a pair of a concrete data state and a location.

An operation $op \in Ops$ can either be an assignment of the form $x := e$ with a variable $x \in X$ and a (side-effect free) arithmetic expression e over variables from X, or an assume operation $[p]$ with a predicate p over variables from X. The semantics of an operation op is defined by the *strongest-postcondition operator* $\mathsf{SP}_{op}(\cdot)$. For a formula ψ and an assignment $x := e$, it is defined as $\mathsf{SP}_{x:=e}(\psi) = \exists \widehat{x} : \psi_{[x \to \widehat{x}]} \wedge (x = e_{[x \to \widehat{x}]})$, and for an assume operation $[p]$ as $\mathsf{SP}_{[p]}(\psi) = \psi \wedge p$. Note that in the implementation we can avoid the existential quantifier in the strongest-postcondition operator for assignments by skolemization.

A *path* $\sigma = \langle (l_i, op_i, l_j), (l_j, op_j, l_k), \ldots, (l_m, op_m, l_n) \rangle$ is a sequence of consecutive edges from G. A path is called *program path* if it starts in the initial location l_{INIT}. The semantics of a path is defined by the iterative application of $\mathsf{SP}_{op}(\cdot)$ for each operation of the path: $\mathsf{SP}_{\sigma}(\psi) = \mathsf{SP}_{op_m}(\ldots(\mathsf{SP}_{op_i}(\psi))\ldots)$. A path σ is called *feasible* if $\mathsf{SP}_{\sigma}(true)$ is satisfiable and *infeasible* otherwise. A location l is called *reachable* if there exists a feasible path from l_{INIT} to l.

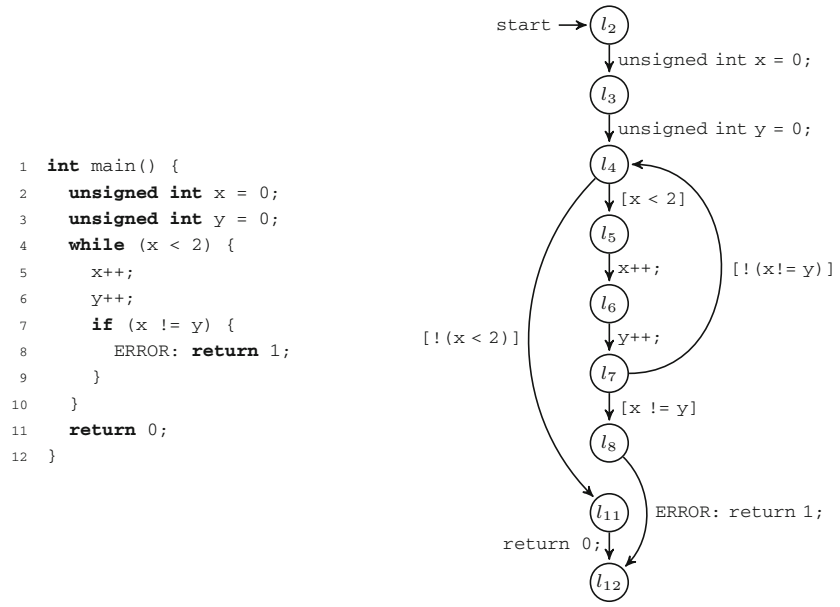
A *verification task* consists of a CFA $A = (L, l_{INIT}, G)$ and an error location $l_{ERR} \in L$, with the goal to show that l_{ERR} is unreachable in A, or to find a feasible error path (i.e., a feasible program path to l_{ERR}) otherwise.

> *Example 1* (*Program and Control-Flow Automaton*) Figure 2 shows an example C program and the corresponding CFA. Location $l_{INIT} = l_2$ is the initial location of this program. The program contains two variables `x` and `y`, which are both initialized to `0`. In the loop of lines 4–10, both variables are incremented as long as `x` is lower than `2`. The CFA nodes corresponding to this loop are l_4, l_5, l_6, and l_7, with l_4 being the loop head. At the end of the loop body in line 7, `x` and `y` are checked for equality. If the variables are not equal, control flows to the error location $l_{ERR} = l_8$ in line 8. We use this CFA as a running example to illustrate the concepts introduced in Sect. 3 and the algorithms presented in Sect. 4.

2.2 Configurable Program Analysis

A configurable program analysis (CPA) [18] specifies the abstract domain that is used for a program analysis. By using the concept of CPAs we can define the abstract domain independently from the analysis algorithm: the CPA algorithm is an algorithm for reachability analysis that can be used with any CPA. Furthermore, CPAs can be combined to compositions of CPAs. The CPAs defined in this work make use of the extension CPA+ (dynamic precision adjustment) [19], but for simplicity we continue to name them CPAs.

A CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ consists of an abstract domain D, a set Π of precisions, a transfer relation $\rightsquigarrow$, and the operators merge, stop, and prec. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, a semilattice $\mathcal{E} = (E, \sqsubseteq)$ over a set E of abstract-domain elements (i.e., abstract states) and a partial order $\sqsubseteq$ (the



(a) Safe program **(b)** Control-flow automaton

Fig. 2 An example C program (**a**) and its CFA (**b**)

join $\sqcup$ of two elements and the join $\top$ of all elements are unique), and a concretization function $[\![\cdot]\!]$ that maps each abstract-domain element to the represented set of concrete states. We call an abstract state $e \in E$ an *abstract error state* if it represents a concrete state at the error location l_{ERR}, i.e., if $\exists c \in (X \to \mathbb{Z}) : (c, l_{ERR}) \in [\![e]\!]$. The transfer relation $\rightsquigarrow \subseteq E \times E \times \Pi$ computes abstract successor states under a precision. The merge operator $\mathsf{merge} : E \times E \times \Pi \to E$ specifies if and how to merge two abstract states when control flow meets under a given precision. The stop operator $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ determines whether an abstract state is covered by a given set of abstract states. The precision-adjustment operator $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ allows adjusting the analysis precision dynamically depending on the current set of reachable abstract states. The operators merge, stop, and prec can be chosen appropriately to influence the abstraction level of the analysis. Common choices include $\mathsf{merge}^{sep}(e, e', \pi) = e'$ (which does not merge abstract states), $\mathsf{stop}^{sep}(e, R, \pi) = (\exists e' \in R : e \sqsubseteq e')$ (which determines coverage by checking whether the given abstract state is less than or equal to any other reachable abstract state according to the semilattice), and $\mathsf{prec}^{id}(e, \pi, \cdot) = (e, \pi)$ (which keeps abstract state and precision unchanged).

2.2.1 CPA Algorithm

CPAs can be used by the CPA algorithm for reachability analysis (cf. Algorithm 1), which gets as input a CPA and an initial abstract state with precision. The algorithm does a classic fixed-point iteration by looping until the set waitlist is empty (all abstract states have been completely processed) and returns the set of reachable abstract states. In each iteration, the algorithm takes one abstract state e with precision π from the waitlist, passes them to the precision-adjustment operator prec, computes all abstract successors, and processes each of the successors. The algorithm checks if there is an existing abstract state with precision in

Algorithm 1 CPA+($\mathbb{D}$, e_{INIT}, π_{INIT}), taken from [19]

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$,
where E denotes the set of elements of the semilattice of D,
and an initial abstract state $e_{INIT} \in E$ with precision $\pi_{INIT} \in \Pi$,
Output: a set of reachable abstract states
Variables: two sets reached and waitlist of elements of $E \times \Pi$

```
1: reached := {(e_INIT, π_INIT)}
2: waitlist := {(e_INIT, π_INIT)}
3: while waitlist ≠ ∅ do
4:    pop (e, π) from waitlist
5:    (ê, π̂) := prec(e, π, reached)                          // Adjust the precision.
6:    for all e' with ê ⇝ (e', π̂) do
7:       for all (e'', π'') ∈ reached do
8:          e_new := merge(e', e'', π̂)                       // Combine with existing abstract state.
9:          if e_new ≠ e'' then
10:            waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')}
11:            reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')}
12:      if not stop(e', {e | (e, ·) ∈ reached}, π̂) then     // Add new abstract state if needed.
13:         waitlist := waitlist ∪ {(e', π̂)}
14:         reached := reached ∪ {(e', π̂)}
15: return {e | (e, ·) ∈ reached}
```

the set of reached states with which the successor abstract state is to be merged (e.g., at join points where control flow meets after completed branching). If this is the case, then the new, merged abstract state with precision substitutes the existing abstract state with precision in both sets reached and waitlist. The stop operator ensures that a new abstract state is inserted into the work sets only if this is needed, i.e., the abstract state is not already covered by an abstract state in the set reached.

2.2.2 Composite CPA

Several CPAs can be combined (Composite pattern) using a *Composite CPA* [18]. The abstract states of the Composite CPA are tuples of one abstract state from each component CPA, the precisions of the Composite CPA are tuples of one precision from each component CPA, and the operators of the Composite CPA delegate to the component CPAs' operators accordingly.

The effect of such a combination of CPAs is that all used CPAs work together in eliminating infeasible paths during the program analysis: one CPA might be able to prove some specific paths infeasible, whereas other CPAs might rule out other infeasible paths. The analysis will only find paths which all used CPAs agree to be feasible. Note that this effect already occurs without any form of communication or information exchange between the component CPAs, and neither does any of the component CPAs need to know anything about the others. However, for an even higher precision, information exchange is possible if desired using the strengthen operator $\downarrow$ [18] and precision-adjustment operator prec [19] of the Composite CPA.

2.2.3 Basic CPAs

The possibility to combine CPAs by using a Composite CPA allows us to separate different concerns: we extract certain common analysis components into separate CPAs and reuse them in flexible combinations with other CPAs, instead of having to redefine them for every analysis from scratch.

For example, for most kinds of program analyses it is necessary to track the program counter, and it is often efficient to track the program counter explicitly rather than symbolically. Thus, we use the *Location CPA* $\mathbb{L}$ [19], which tracks exactly the program counter (with a flat lattice over all program locations, a constant precision, and the operators merge^{sep}, stop^{sep}, and prec^{id}), and we use this CPA in addition to other CPAs whenever explicit tracking of the program counter is necessary.

Furthermore, in order to track the abstract reachability graph (ARG) over the abstract states in the (flat) set $\mathsf{reached}$, we define an additional *ARG CPA* $\mathbb{A}$, which stores the predecessor–successor relationship between abstract states. The ARG CPA allows us to reconstruct abstract paths in the ARG: An *abstract path* is a sequence $\langle e_0, \ldots, e_n \rangle$ of abstract states such that for any pair (e_i, e_{i+1}) with $i \in \{0, \ldots, n-1\}$ either e_{i+1} is an abstract successor of e_i, or e_{i+1} is the result of merging an abstract successor of e_i with some other abstract state(s). If both the Location CPA and the ARG CPA are used, we can reconstruct from an abstract path the path that it represents in the CFA.

2.3 Counterexample-Guided Abstraction Refinement (CEGAR)

Counterexample-guided abstraction refinement (CEGAR) [34] is an approach for iteratively finding an analysis precision that is strong enough to prove the program safe and coarse enough to allow for an efficient analysis. Starting with a coarse initial precision (typically an empty set of facts, e.g., predicates), an abstract model that is an overapproximation of the program is created by the underlying reachability analysis. If an abstract state that belongs to the error location is found in the abstract model, the concrete program path that leads to this state is reconstructed from the ARG and checked for feasibility. If the error path is feasible, the program is unsafe and the analysis terminates. Otherwise, the error path is infeasible, and we refine the precision of the analysis to be precise enough to eliminate this infeasible error path from the ARG. Then the analysis is restarted, and the steps are repeated until either a concrete error path is found, or the abstract model (and thus the program) is proven safe.

CEGAR is often combined with lazy abstraction [51], which makes this approach more efficient by increasing the precision only selectively in parts of the state space where it is needed and by not restarting the analysis from scratch after each refinement. We use the CPA algorithm for the creation of the abstract model in the CEGAR approach and let the refinement influence the precision of the used CPA(s).

3 Predicate CPA

Our goal is to define a configurable and flexible framework for predicate-based approaches that is helpful both in theory (by simplifying development and studying of approaches) as well as in practice (by being customizable for different use cases). In addition, a mature and efficient implementation of this framework should allow reliable scientific experiments and application in practice of the approaches that are integrated now or in the future.

The core of our framework is defined as a CPA for predicate-based analyses, which we name the *Predicate CPA* $\mathbb{P}$. It is an extension of an existing CPA for predicate abstraction with adjustable-block encoding (ABE) [21], and a preliminary version was already published [25]. The Predicate CPA $\mathbb{P} = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, \mathsf{merge}_{\mathbb{P}}, \mathsf{stop}_{\mathbb{P}}, \mathsf{prec}_{\mathbb{P}})$ consists of the abstract domain $D_{\mathbb{P}}$, the set $\Pi_{\mathbb{P}}$ of precisions, the transfer relation $\rightsquigarrow_{\mathbb{P}}$, the merge operator $\mathsf{merge}_{\mathbb{P}}$, the stop operator $\mathsf{stop}_{\mathbb{P}}$, and the operator $\mathsf{prec}_{\mathbb{P}}$ for dynamic precision adjustment. Additionally, we will define an operator $\mathsf{fcover}_{\mathbb{P}}$ for IMPACT-style forced cov-

ering and an operator $\mathsf{refine}_{\mathbb{P}}$ for refinements. In the following, we will define and describe these parts in more details. We also provide an extended version of the CPA algorithm, and in the next section we will describe how to express various algorithms for software verification using the concepts defined here. The examples in this section illustrate some cases that occur when verifying the running example program given in Fig. 2 using one of these algorithms from Sect. 4.

3.1 Abstract Domain, Precisions, and CPA Operators

The abstract domain $D_{\mathbb{P}} = (C, \mathcal{E}_{\mathbb{P}}, [\![\cdot]\!]_{\mathbb{P}})$ consists of the set C of concrete states, the semilattice $\mathcal{E}_{\mathbb{P}}$ over abstract states, and the concretization function $[\![\cdot]\!]_{\mathbb{P}}$. The semilattice $\mathcal{E}_{\mathbb{P}} = (E_{\mathbb{P}}, \sqsubseteq_{\mathbb{P}})$ consists of the set $E_{\mathbb{P}}$ of abstract states and the partial order $\sqsubseteq_{\mathbb{P}}$.

3.1.1 Abstract States

Because of the use of adjustable-block encoding [21], an abstract state $e \in E_{\mathbb{P}}$ of the Predicate CPA is a triple $(\psi, l^{\psi}, \varphi)$ of an abstraction formula ψ, the abstraction location l^{ψ} (the program location where ψ was computed), and a path formula φ. Both formulas are first-order formulas over predicates over the program variables from the set X, and an abstract state represents all concrete states that satisfy their conjunction: $[\![(\psi, l^{\psi}, \varphi)]\!]_{\mathbb{P}} = \{(c, \cdot) \in C \mid c \models (\psi \wedge \varphi)\}$. The partial order $\sqsubseteq_{\mathbb{P}}$ is defined as $(\psi_1, l^{\psi}{}_1, \varphi_1) \sqsubseteq_{\mathbb{P}} (\psi_2, l^{\psi}{}_2, \varphi_2) = ((\psi_1 \wedge \varphi_1) \Rightarrow (\psi_2 \wedge \varphi_2))$, i.e., an abstract state is less than or equal to another state if the conjunction of the formulas of the first state implies the conjunction of the formulas of the other state. Abstract states where the path formula φ is *true* are called *abstraction states*, other abstract states are *intermediate states*. The transfer relation produces only intermediate states, and at the end of a block of program operations the operator prec computes an abstraction state from an intermediate state. The initial abstract state is the abstraction state $(true, l_{INIT}, true)$.

The path formula of an abstract state is always represented syntactically as an SMT formula. The representation of the abstraction formula, however, can be configured. We can either use a binary-decision diagram (BDD) [31], as in classic predicate abstraction [15,47], or an SMT formula similar to the path formula. Using BDDs allows performing cheap entailment checks between abstraction states at the cost of an increased effort for constructing the BDDs.

3.1.2 Precisions

A precision $\pi \in \Pi_{\mathbb{P}}$ of the Predicate CPA is a mapping from program locations to sets of predicates over the program variables. This allows using a different abstraction level at each location in the program (lazy abstraction). The initial precision is typically the mapping $\pi(l) = \emptyset$, for all $l \in L$. The Predicate CPA does not use dynamic precision adjustment [19] during an execution of the CPA algorithm: instead the precision is adjusted only during a refinement step, if the predicate refinement strategy is used. The only operation that changes its behavior based on the precision is the predicate abstraction that may be computed at block ends by the operator $\mathsf{prec}_{\mathbb{P}}$.

3.1.3 Transfer Relation

The transfer relation $(\psi, l^{\psi}, \varphi) \rightsquigarrow ((\psi, l^{\psi}, \varphi'), \pi)$ for a CFA edge (l_i, op_i, l_j) produces a successor state $(\psi, l^{\psi}, \varphi')$ such that the abstraction formula and location stay unchanged and the path formula φ' is created by applying the strongest-postcondition operator for the current

CFA edge to the previous path formula: $\varphi' = \mathsf{SP}_{op_i}(\varphi)$. Note that this is an inexpensive, purely syntactical operation that does not involve any actual solving, and that it is a precise operation, i.e., it does not perform any form of abstraction.

3.1.4 Merge Operator

The merge operator $\mathsf{merge}_{\mathbb{P}}$ combines intermediate states that belong to the same block (their abstraction formula and location is the same) and keeps any other abstract states separate:

$$\mathsf{merge}_{\mathbb{P}}\left(\left(\psi_1, l^{\psi}{}_1, \varphi_1\right), \left(\psi_2, l^{\psi}{}_2, \varphi_2\right), \pi\right) = \begin{cases} \left(\psi_2, l^{\psi}{}_2, \varphi_1 \vee \varphi_2\right) & \text{if } \left(\psi_1 = \psi_2\right) \wedge \left(l^{\psi}{}_1 = l^{\psi}{}_2\right) \\ \left(\psi_2, l^{\psi}{}_2, \varphi_2\right) & \text{otherwise} \end{cases}$$

This definition is common for analyses based on adjustable-block encoding (ABE) [21]. By merging abstract states inside each block, the number of abstract states in the ARG is kept small, and no precision is lost due to merging, because the path formula of an abstract state exactly represents the path(s) from the block start without abstraction. At the same time the loss of information that would lead to a path-insensitive analysis if states would be merged across blocks is avoided. The result is that the ARG, if projected to contain only abstraction states, forms an abstract-reachability tree (ART) like in a path-sensitive analysis without ABE. This is necessary for being able to reconstruct abstract paths, for example during refinement and for reporting concrete error paths.

3.1.5 Stop Operator

The stop operator $\mathsf{stop}_{\mathbb{P}}$ checks coverage only for abstraction states and always returns *false* for intermediate states:

$$\mathsf{stop}_{\mathbb{P}}((\psi, l^{\psi}, \varphi), R, \pi) = \begin{cases} \exists (\psi', l^{\psi\,\prime}, \varphi') \in R : \varphi' = \mathit{true} \wedge (\psi, l^{\psi}, \varphi) \sqsubseteq_{\mathbb{P}} (\psi', l^{\psi\,\prime}, \varphi') & \text{if } \varphi = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Because the path formula of an abstraction state is always *true*, the first case is equivalent to checking if there exists an abstraction state $(\psi', \cdot, \mathit{true})$ in the set R whose abstraction formula ψ' is implied by the abstraction formula ψ of the current abstraction state $(\psi, l^{\psi}, \varphi)$. If abstraction formulas are represented by BDDs, this is an efficient operation, otherwise a potentially costly SMT query is required. The coverage check for intermediate states is omitted for efficiency, because it would always need to involve (potentially many) SMT queries. Note that this implies that infinitely long sequences of intermediate states must be avoided, otherwise the analysis would not terminate.

3.1.6 Precision-Adjustment Operator

The precision-adjustment operator $\mathsf{prec}_{\mathbb{P}}$ either returns the input abstract state and precision, or converts an intermediate state into an abstraction state performing predicate abstraction. The decision is made by the block-adjustment operator blk [21], which returns *true* or *false* depending on whether the current block ends at the current abstract state and thus an abstraction should be computed. The decision can be based on the current abstract state as well as on information about the current program location. We define the following common choices

for blk: blk^{lf} returns *true* at loop heads, function calls/returns, and at the error location l_{ERR}, leading to a behavior similar to large-block encoding (LBE) [11]. blk^{l} returns *true* only at loop heads and at the error location l_{ERR}. The abstraction at the error location is needed for detecting the reachability of abstract error states due to the satisfiability check that is implicitly done by the abstraction computation if the precision is not empty. blk^{never} always returns *false*. This will prevent all abstractions and (due to how $\mathsf{stop}_{\mathbb{P}}$ is defined) also prevents coverage between abstract states. This means that an analysis with blk^{never} will unroll the CFA endlessly until other reasons prevent this. We will show a meaningful application of blk^{never} in Sect. 4.1 (BMC).

The boolean predicate abstraction [7] $(\varphi)^{\rho}_{\mathbb{B}}$ of a formula φ for a set ρ of predicates is the strongest boolean combination of predicates from ρ that is implied by φ. It can be computed using an SMT solver by solving $\varphi \wedge \bigwedge_{p_i \in \rho}(v_{p_i} \Leftrightarrow p_i)$ and enumerating all its models with respect to the fresh boolean variables $v_{p_1}, \ldots, v_{p_{|\rho|}}$. For each model we create a conjunction over the predicates from ρ, with each predicate p_i being negated if the model maps the corresponding variable v_{p_i} to *false*. The result of $(\varphi)^{\rho}_{\mathbb{B}}$ is the disjunction of all these conjunctions. To create an abstraction state from an intermediate state $(\psi, l^{\psi}, \varphi)$ at program location l (which is tracked by another CPA that runs in parallel to the Predicate CPA as a sibling component within the same Composite CPA and from which the location can be retrieved), we compute the boolean predicate abstraction $(\psi \wedge \varphi)^{\pi(l)}_{\mathbb{B}}$ for the formula $\psi \wedge \varphi$ and the set $\pi(l)$ of predicates from the precision, after adjusting the variable names of ψ to match those of φ (because the variables from ψ need to match the 'oldest' variables in φ). Thus, we can define the precision-adjustment operator as

$$\mathsf{prec}_{\mathbb{P}}\left(\left(\psi, l^{\psi}, \varphi\right), \pi, R\right) = \begin{cases} \left(\left((\psi \wedge \varphi)^{\pi(l)}_{\mathbb{B}}, l, \mathit{true}\right), \pi\right) & \text{if } \mathsf{blk}\left(\left(\psi, l^{\psi}, \varphi\right), l\right) \\ \left(\left(\psi, l^{\psi}, \varphi\right), \pi\right) & \text{otherwise} \end{cases}$$

Note that, if an abstraction is going to be computed, the current path formula φ precisely represents all the paths within this block (i.e., from the last abstraction state to the current abstract state). Thus, we name this path formula the *block formula* for the block ending in the current abstract state. If the precision is empty for the current program location, the outcome of the abstraction computation will always simply be *true* and no SMT queries are necessary. If the precision for the current program location is {*false*}, the abstraction computation will be equivalent to a simple satisfiability check, and the outcome will always be either *true* or *false*.

> *Example 2* (*Boolean Predicate Abstraction*) Given an intermediate abstract state $(\psi, l^{\psi}, \varphi)$ with
>
> $\psi = (x = y)$, which is rewritten to $x_0 = y_0$, and
> $\varphi = x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1$,
>
> and a set ρ of predicates with $\rho = \{x = y\}$ we introduce a boolean variable $v_{x=y}$ for the (instantiated) predicate $x_1 = y_1$ and use an SMT solver to enumerate all models of the following formula
>
> $$x_0 = y_0 \wedge x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge (v_{x=y} \Leftrightarrow x_1 = y_1)$$
>
> with respect to variable $v_{x=y}$. In this case, $\{v_{x=y} \mapsto \mathit{true}\}$ is the only such model. As a result, the abstraction is $x = y$.

3.2 Refinement

The refinement operator $\mathsf{refine}_{\mathbb{P}}$ takes as input two sets $\mathsf{reached} \subseteq E \times \Pi$ of reached abstract states and $\mathsf{waitlist} \subseteq E \times \Pi$ of frontier abstract states and expects $\mathsf{reached}$ to contain an abstract error state at error location l_{ERR} that represents a specification violation. refine either returns the sets unchanged (if the abstract error state is reachable, i.e., there is a feasible error path), or modified such that the sets can be used for continuing the state-space exploration with an increased precision (if the error path is infeasible). The operator works in four steps.

3.2.1 Abstract-Counterexample Construction

The first step is to construct the set of abstract paths between the initial abstract state and the abstract error state. Traditionally, in an abstract reachability tree, there would exist exactly one such abstract path. Because we use ABE, however, intermediate states can be merged, and thus the abstract states form an abstract reachability *graph*, where several paths can exist from the initial abstract state to the abstract error state. All these abstract paths to the abstract error state contain the same sequence of abstraction states with varying sequences of intermediate states in between. This is due to the fact that abstraction states are never merged, and intermediate states are merged only locally within a block. Thus, the ARG, if projected to the abstraction states, still forms a tree. The initial abstract state is always an abstraction state by definition, and our choices of the block-adjustment operator blk ensure that all abstract error states are also abstraction states. Thus, we define as *abstract counterexample* the sequence $\langle e_0, \ldots, e_n \rangle$ that begins with the initial abstract state ($e_0 = e_{INIT}$), ends with the abstract error state e_n, and contains all abstraction states $e_1, \ldots, e_{n-1}$ on paths between these two abstract states. This sequence can be reconstructed from the ARG by following a single arbitrary abstract path backwards from the abstract error state (using the information tracked by the ARG CPA), without needing to explicitly enumerate all (potentially exponentially many) abstract paths between the initial abstract state and the abstract error state.

3.2.2 Feasibility Check

From an abstract counterexample $\langle e_0, \ldots, e_n \rangle$ we can create a sequence $\langle \varphi_1, \ldots, \varphi_n \rangle$ of block formulas where each φ_i represents all paths between e_{i-1} and e_i. Note that each φ_i is also exactly the same formula as the path formula that was used as input when computing the abstraction for state e_i. Then we check whether there exists a feasible concrete path that is represented by one of the abstract paths of the abstract counterexample by checking the *counterexample formula* $\bigwedge_{i=1}^{n} \varphi_i$ for satisfiability in a single SMT query. If satisfiable, the analysis has found a violation of the specification and terminates. Otherwise, i.e., if all abstract paths to the abstract error state are infeasible under the concrete program semantics, we say that the abstract counterexample is spurious, and a refinement of the abstract model is necessary to eliminate this infeasible error path from the ARG.

3.2.3 Interpolation

To refine the abstract model, $\mathsf{refine}_{\mathbb{P}}$ uses Craig interpolation [38] to discover abstract facts that allow eliminating the infeasible error path. Given a sequence $\widehat{\varphi} = \langle \varphi_1, \ldots, \varphi_n \rangle$ of formulas whose conjunction is unsatisfiable, a sequence $\langle \tau_0, \ldots, \tau_n \rangle$ is an inductive sequence of interpolants for $\widehat{\varphi}$ if

1. $\tau_0 = \mathit{true}$ and $\tau_n = \mathit{false}$,
2. $\forall i \in \{1, \ldots, n\} : \tau_{i-1} \wedge \varphi_i \Rightarrow \tau_i$, and
3. for all $i \in \{1, \ldots, n-1\}$, τ_i references only variables that occur in $\bigwedge_{j=1}^{i} \varphi_i$ as well as in $\bigwedge_{j=i+1}^{n} \varphi_i$.

Note that every interpolation sequence starts with no assumption ($\tau_0 = \mathit{true}$) and ends with a contradiction ($\tau_n = \mathit{false}$), and that $\tau_i \Rightarrow \neg \bigwedge_{j=i+1}^{n} \varphi_j$ follows from the definition, for all $i \in \{1, \ldots, n\}$. For many common SMT theories, interpolants are guaranteed to exist and can be computed using off-the-shelf SMT solvers from a proof of unsatisfiability for $\bigwedge_{i=1}^{n} \varphi_i$. Note that in general there exist many possible sequences of interpolants for a single infeasible error path.

Example 3 (*Interpolation*) Given a sequence $\widehat{\varphi} = \langle \varphi_1, \varphi_2 \rangle$ of formulas, where

$\varphi_1 = (x_0 = 0 \wedge y_0 = 0)$ and
$\varphi_2 = (x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge (x_1 \neq y_1))$,

the sequence $\langle \tau_0, \tau_1, \tau_2 \rangle$ with

$\tau_0 = \mathit{true}$,
$\tau_1 = (x_0 = y_0)$, and
$\tau_2 = \mathit{false}$

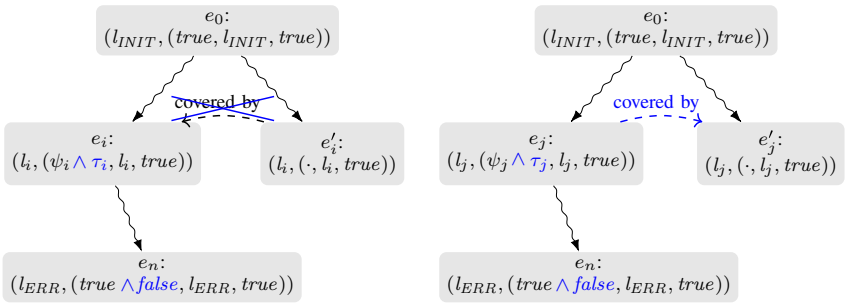
is a valid sequence of interpolants for $\widehat{\varphi}$, because it satisfies the definition above:

1. $\tau_0 = \mathit{true}$ and $\tau_n = \mathit{false}$,
2. $\mathit{true} \wedge x_0 = 0 \wedge y_0 = 0 \Rightarrow x_0 = y_0$ and
   $x_0 = y_0 \wedge x_0 < 2 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge (x_1 \neq y_1) \Rightarrow \mathit{false}$, and
3. τ_1 references the variables x_0 and y_0, which occur in both φ_1 and φ_2.

3.2.4 Refinement Strategies

Lastly, $\mathsf{refine}_{\mathbb{P}}$ needs to refine the precision of the analysis such that afterwards the analysis is guaranteed to not encounter the same error path again. A refinement strategy uses the current spurious abstract counterexample $\langle e_0, \ldots, e_n \rangle$ and the corresponding sequence $\langle \tau_0, \ldots, \tau_n \rangle$ of interpolants to modify the sets reached and waitlist. For this step, two common approaches exist. Afterwards, the refinement is finished, the modified sets reached and waitlist are returned to the analysis, and the analysis continues with building the abstract model (which will now be more precise).

IMPACT *Refinement*. One refinement strategy is to perform a refinement similar to the function REFINE of the IMPACT algorithm [61]. The IMPACT refinement strategy takes each abstraction state ψ_i of the abstract counterexample and conjoins to its abstraction formula the corresponding interpolant τ_i. If an abstract state is actually strengthened by this (i.e., the previous abstraction formula did not already imply the interpolant), we also need to recheck all coverage relations of this abstract state. Figure 3a outlines such a situation: an abstract state e'_i previously covered by another abstract state e_i is now no longer covered, because the abstraction formula of e_i was strengthened by the refinement. In this case, we uncover and readd all leaf abstract states in the subgraph of the ARG that starts with the uncovered abstract state e'_i to the set waitlist. We also check for each of the strengthened abstract states whether



(a) Abstract state e'_i may lose coverage by strengthened abstract state e_i

(b) Strengthened abstract state e_j may gain coverage by abstract state e'_j

Fig. 3 Sketches of the process of rechecking coverage relations after IMPACT refinement. Squiggly arrows represent paths between abstraction states and hide intermediate states

it is now covered by any other abstract state at the same program location. If this is successful, i.e., if a strengthened abstract state e_j is now covered by another abstract state e'_j as shown in Fig. 3b, we mark the subgraph that starts with that strengthened abstract state e_j as covered and remove all leafs therein from waitlist (we do not need to expand covered abstract states). The only change to the set reached is the removal of all abstract states whose abstraction formula is now equivalent to *false* and their successors. Due to the properties of interpolants, this is guaranteed to be the case for at least the abstract error state.

Example 4 (IMPACT *Refinement*) Given

- a set of program locations $L = \{l_2, l_4, l_8\}$,
- an abstract counterexample $\langle e_0, e_1, e_2 \rangle$,
- a corresponding sequence of program locations $\langle l_2, l_4, l_8 \rangle$ where

  e_0 is at program location $l_2 = l_{INIT}$,
  e_1 is at program location l_4, and
  e_2 is at program location $l_8 = l_{ERR}$,

- and a sequence of interpolants $\langle \tau_0, \tau_1, \tau_2 \rangle$ with

  $\tau_0 = true$ at l_2,
  $\tau_1 = (x_0 = y_0)$ at l_4, and
  $\tau_2 = false$ at l_8,

we directly strengthen the abstract states e_1 and e_2 by conjoining the interpolant $x_0 = y_0$ to the abstraction formula of e_1 and conjoining the interpolant *false* to the abstraction formula of e_2.

We then remove e_2 from the set reached because its abstraction formula is now equivalent to *false*, check if the strengthening of the abstraction formula of e_1 invalidated any coverage relations, such that we readd leafs of subgraphs of abstract states that became uncovered, check if the strengthening of the abstraction formula of e_1 caused e_1 to become covered by any other state so that we remove all leaf states of its subgraph from the set waitlist, and then continue the state-space exploration.

Predicate Refinement. Another refinement strategy is used for traditional lazy predicate abstraction. It extracts the atoms of the interpolants as predicates, creates a new precision π with these predicates, and restarts (a part of) the analysis with a new precision that is extended by π.

The precision π is a mapping from program locations to sets of predicates, and we add predicates to the precision only for program locations where they are necessary. Assuming that, starting from an abstract counterexample $\langle e_0, \ldots, e_n \rangle$ with abstraction states at program locations $\langle l_0, \ldots, l_n \rangle$ we obtained a sequence $\langle \tau_0, \ldots, \tau_n \rangle$ of interpolants and extracted a sequence $\langle \rho_0, \ldots, \rho_n \rangle$ of sets of predicates. Then we add each predicate to the precision for the program location that corresponds to the point in the abstract counterexample where the predicate appears in the interpolant, i.e., $\pi(l) = \bigcup_{i=0}^{n} (\rho_i \text{ if } l = l_i \text{ else } \emptyset)$. Note that due to the properties of interpolants, $\pi(l_{ERR})$ will always be $\{false\}$. We take the precision π with the new predicates and the existing precision π_n that is associated in the set reached with the abstract error state e_n and join them element-wise to create the new precision π' with $\forall l \in L : \pi'(l) = \pi_n(l) \cup \pi(l)$ that will be used in the subsequent analysis.

Finally, the sets reached and waitlist are prepared for continuing with the analysis. We remove only those parts of the ARG for which the new predicates are necessary. For this, we determine the first abstract state of the abstract counterexample for which the new precision π' would lead to more predicates being used in the abstraction computation than the originally used predicates and call this the pivot abstract state. Then we remove the subgraph of the ARG that starts with the pivot abstract state from the sets reached and waitlist, as well as all abstract states that were covered by one of the removed abstract states. To ensure that the removed parts of the ARG get re-explored, we take all remaining parents of removed abstract states, replace the precision with which they are associated in reached with the new precision π', and add them to the set waitlist. This has not only the effect of avoiding the re-exploration of unchanged parts of the ARG, but also leads to the new predicates being used only in the relevant part of the ARG, with other parts of the program state space being explored with different (possibly more abstract and thus more efficient) precisions.

Example 5 (*Predicate Refinement*) Given

- a set of program locations $L = \{l_2, l_4, l_8\}$,
- an initial precision π_n, with $\forall l \in L : \pi_n(l) = \emptyset$,
- an abstract counterexample $\langle e_0, e_1, e_2 \rangle$,
- a corresponding sequence of program locations $\langle l_2, l_4, l_8 \rangle$ where

  e_0 is at program location $l_2 = l_{INIT}$,
  e_1 is at program location l_4, and
  e_2 is at program location $l_8 = l_{ERR}$,

- and a sequence of interpolants $\langle \tau_0, \tau_1, \tau_2 \rangle$ with

  $\tau_0 = true$ at l_2,
  $\tau_1 = (x_0 = y_0)$ at l_4 and
  $\tau_2 = false$ at l_8,

we extract the sequence $\langle \rho_0, \rho_1, \rho_2 \rangle$ of sets of predicates with

$\rho_0 = \{\}$ at l_2,
$\rho_1 = \{x = y\}$ at l_4, and
$\rho_2 = \{false\}$ at l_8.

We then use this sequence of sets of predicates to construct the precision π:

$\pi(l_2) = \rho_0 = \{\}$,
$\pi(l_4) = \rho_1 = \{x = y\}$, and
$\pi(l_8) = \rho_2 = \{false\}$.

Joining the previous precision π_n with the newly obtained precision π yields the updated precision π' ($\pi' = \pi$ because π_n is empty for each location).

As a result, the first abstract state in the abstract counterexample that is affected by the new precision is e_1, which therefore becomes the pivot state and is removed from the ARG, along with all its descendants in the ARG, including e_2. Then, starting from the predecessors of e_1, the state space is re-explored using the new precision π'.

3.3 Forced Covering

Forced coverings were introduced for lazy abstraction with interpolants (IMPACT) [61] for a faster convergence of the analysis. Typically, when the CPA algorithm creates a new successor abstract state for an IMPACT analysis, this new abstract state is too abstract to be covered by existing abstract states, since the IMPACT refinement strategy is used, which leads to all new abstraction states being equivalent to *true*. If an abstract state cannot be covered, the analysis needs to further create successors of it, leading to more abstract states and possibly more refinements. The idea of forced covering is to strengthen new abstract states such that they are covered by existing abstract states immediately if possible.

We define an operator $\mathsf{fcover}_{\mathbb{P}} : 2^{E \times \Pi} \times E \times \Pi \to 2^{E \times \Pi}$ that takes as input the set $\mathsf{reached}$ of reachable abstract states and an abstract state e with precision π and returns an updated set $\mathsf{reached}'$ of reachable abstract states. The operator may replace e and other abstract states in $\mathsf{reached}$ with strengthened versions, if it can guarantee that this is sound and if afterwards the strengthened version of e is covered by another abstract state in $\mathsf{reached}'$. A trivial implementation of this operator is $\mathsf{fcover}^{id}(\mathsf{reached}, e, \pi) = \mathsf{reached}$, which does not strengthen abstract states and returns the set $\mathsf{reached}$ unchanged.

An alternative implementation is $\mathsf{fcover}^{\text{IMPACT}}$, which adopts the strategy for forced coverings presented for lazy abstraction with interpolants [61]. We extend this approach here to support adjustable-block encoding. Because the Predicate CPA does not attempt to cover intermediate states (only abstraction states), we also only attempt forced coverings for abstraction states. Figure 4 shows a sketch of the concept of forced covering in IMPACT to help visualize the following explanation: Given an abstraction state e that should be covered if possible, the candidate abstract states for covering are those abstraction states that belong to the same location, were created before e, and are not covered themselves. For each candidate e', we first determine the nearest common ancestor abstraction state $\hat{e}$ of e and e' (using the information tracked by the ARG CPA). Now let us denote the abstraction formulas of e' and $\hat{e}$ with ψ' and $\hat{\psi}$, respectively, and let φ be the path formula that represents the paths from $\hat{e}$ to e. We then determine whether ψ' also holds for e by checking if $\hat{\psi} \wedge \varphi \implies \psi'$ holds, i.e., whether it is impossible to reach a concrete state that is not represented by ψ' when starting at $\hat{e}$ and following the paths to e. If this holds, we can strengthen the abstraction formula

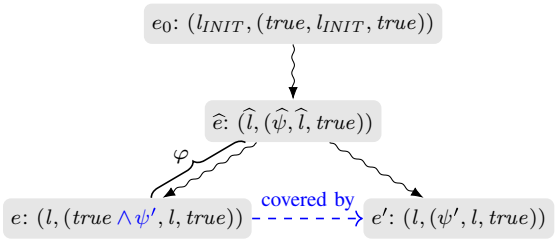


Fig. 4 Concept sketch for fcover$^{\text{IMPACT}}$, blue parts are added on successful forced covering, i.e., if $\hat{\psi} \wedge \varphi \Rightarrow \psi'$

of e with ψ' (which immediately lets us cover e by e'). Furthermore, if there are abstraction states along the paths from $\hat{e}$ to e, we need to strengthen these states, too, in order to keep the ARG well-formed. We can do so by computing interpolants at the appropriate locations along the paths for the query that we have just solved and strengthen the abstract states with the interpolants. If the query does not hold, we switch to the next candidate abstract state and try again. Finally, fcover$^{\text{IMPACT}}$ returns an updated set reached with strengthened abstract states, or the original set reached if forced covering was unsuccessful for each of the candidates. Note that this forced-covering strategy is similar to interpolation-based refinement with the IMPACT refinement strategy, just that we attempt to prove that ψ' instead of *false* holds at the end of the path, and that the refined path does not start at the initial abstract state but at $\hat{e}$.

3.4 An Extended CPA Algorithm

In order to be able to use all the features of the Predicate CPA and support approaches such as lazy abstraction, we also need to slightly extend the CPA algorithm. The extended version, which we call the CPA++ algorithm, is shown as Algorithm 2. Compared to the original version (Algorithm 1), it has the following differences:

1. CPA++ gets reached and waitlist as input and returns updated versions of both of them, instead of getting an initial abstract state and returning a set of reachable abstract states.
2. CPA++ calls a function abort to determine whether it should abort early for each found abstract state (lines 16–17).
3. CPA++ calls the precision-adjustment operator immediately for each new abstract state (line 7) instead of only before expanding an abstract state.
4. CPA++ attempts a forced covering by calling fcover before expanding an abstract state (lines 3–5).

The first two changes allow calling CPA++ iteratively and keep expanding the same set of abstract states, which is necessary for CEGAR with lazy abstraction (where we want to abort as soon as we find an abstract error state and continue after refinement without restarting from scratch; abort(e) is typically implemented to return *true* if e is an abstract state at error location l_{ERR}). The new position of the call to the precision-adjustment operator is necessary because previously the resulting abstract states ($\widehat{e}$ in Algorithm 1) were never put into reached. However, we need the abstract states resulting from prec to be in reached, because among them are the abstraction states of the Predicate CPA, which are necessary for refinement.

Similar changes to the CPA algorithm have been used previously [22,25]; we now combine them in order to provide an all-encompassing algorithm for reachability that we can use as building block for our unifying framework for predicate-based software verification.

Algorithm 2 CPA++($\mathbb{D}$, reached, waitlist, abort), extension of Algorithm 1

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ with additional operator fcover,
where E denotes the set of elements of the semilattice of D,
a set $\mathsf{reached} \in E \times \Pi$ of reachable abstract states
a set $\mathsf{waitlist} \in E \times \Pi$ of frontier abstract states, and
a function $\mathsf{abort} : E \to \mathbb{B}$ that defines whether the algorithm should abort early
Output: the updated sets reached and waitlist

```
1: while waitlist ≠ ∅ do
2:    pop (e, π) from waitlist
3:    reached := fcover(reached, e, π)
4:    if (e, π) ∉ reached then
5:       continue                                   // Forced covering was successful.
6:    for all e' with e ⇝ (e', π) do
7:       (ê, π̂) := prec(e', π, reached)             // Adjust the precision of the abstract state.
8:       for all (e'', π'') ∈ reached do
9:          e_new := merge(ê, e'', π̂)               // Combine with existing abstract state.
10:         if e_new ≠ e'' then
11:            waitlist := (waitlist ∪ {(e_new, π̂)}) \ {(e'', π'')}
12:            reached := (reached ∪ {(e_new, π̂)}) \ {(e'', π'')}
13:      if not stop(ê, {e | (e, ·) ∈ reached}, π̂) then   // Add new abstract state if needed.
14:         waitlist := waitlist ∪ {(ê, π̂)}
15:         reached := reached ∪ {(ê, π̂)}
16:         if abort(ê) then
17:            return (reached, waitlist)
18: return (reached, waitlist)
```

4 Unifying SMT-Based Approaches for Software Verification

In this section, we will give a unifying overview of four widely used approaches to software verification: bounded model checking (BMC), k-induction, predicate abstraction, and the IMPACT algorithm. We reformulate the approaches in our theoretical framework from the previous section and illustrate their differences using our example program.

In the following, the Predicate CPA $\mathbb{P}$ is always combined with at least the CPA $\mathbb{L}$ for program-counter tracking and the ARG CPA $\mathbb{A}$ for tracking the predecessor–successor as well as coverage relations between ARG nodes. We show relations between ARG nodes graphically in the figures and omit them for ease of presentation when notating abstract states as tuples. For path formulas, we use a skolemized notation based on SSA indices [39], which is easier to read than existential quantification of many variables. Index addition and removal is done implicitly when converting between abstraction formulas and path formulas.

4.1 Bounded Model Checking

For bounded model checking, we set the ABE block size to infinite (we call this whole-program encoding) by using the block operator blk^{never}, and we use fcover^{id} (i.e., no forced coverings). Additionally, we combine the Predicate CPA with a CPA for bounding the state space besides the typical basic CPAs.

The *Loop-Bound CPA* $\mathbb{LB}$ tracks in its abstract states for every loop of the program how often the loop body was traversed on the current program path. It associates each loop-head location with a counter that starts with -1 and is incremented by the transfer relation whenever the respective location is reached. The precision is the loop bound k: $\pi = k$, with $k > 0$. The transfer relation of the Loop-Bound CPA is unsound on purpose: it does not produce any successor abstract states for abstract states in which one of the counters for the loop-head

locations is equal to the loop bound k in the precision and thus prevents the analysis from exploring any paths for more than k loop iterations. Apart from that, the Loop-Bound CPA uses the standard operators merge^{sep}, stop^{sep}, and prec^{id}.

This configuration leads to an analysis without abstraction computations, expensive coverage checks, and refinements. Instead, the CPA++ algorithm simply unrolls the CFA (within the loop bound), and each abstract state contains a path formula that exactly represents the paths from the initial location to this abstract state. We wrap the CPA++ algorithm in another algorithm that checks satisfiability of the path formula of each abstract error state after the CPA++ algorithm has finished (we can use Algorithm 3, which is discussed in Sect. 4.2, for this by omitting lines 15–23). If at least one path formula is satisfiable (for efficiency, we check the disjunction of all path formulas at once in line 10 of Algorithm 3), then there exists a feasible path to the error location, i.e., the specification is violated.

We can also implement a forward-condition check [44] by making an additional SMT query for the satisfiability of the path formulas of all those abstract states for which the Loop-Bound CPA has unsoundly restricted the successor abstract states. If none of these path formulas is satisfiable, the specification is proven to hold for the program. If for a given loop bound k the result was inconclusive (i.e., no specification violation found but the forward-condition check was unsuccessful, too), we can repeat the bounded model check with a higher k.

> *Example 6* (*BMC*) If we apply BMC with $k = 1$ to the program of Fig. 2, unrolling the CFA yields the ARG depicted in Fig. 5. In this figure, each abstract state is a tuple $(l, (\psi, l^{\psi}, \varphi), \{l_4 \mapsto i\})$ of the abstract states of $\mathbb{L}$, $\mathbb{P}$, and $\mathbb{LB}$. The path formula of the abstract state e_8, which is the only abstract state at error location $l_{ERR} = l_8$, is unsatisfiable. Therefore, no bug is reachable within one loop unrolling. The abstract state e_{10} is the last state in this ARG because here the bound $k = 1$ is reached. In order to do a forward-condition check we check the satisfiability of the path formula of e_{10}. Because the formula is satisfiable and thus, e_{10} is reachable, we can conclude that the bound $k = 1$ is not large enough to fully verify this program.

4.2 k-Induction

For ease of presentation, we assume here that the loop head is not reachable from the error location l_{ERR} and that the analyzed program has exactly one loop whose loop-head location is l_{LH}. In practice, k-induction can be applied to programs with many loops [13].

k-Induction, like BMC, is an approach that at its core does not rely on abstraction techniques. We present an algorithm for k-induction-based verification based on the Predicate CPA as Algorithm 3. This algorithm supports iterative deepening and injection of continuously refined invariants. We can use this algorithm in combination with (external) standard invariant-generation techniques, such as data-flow analysis [57,63] and template-based approaches [16,36]. This is necessary, because often the safety property of a verification task is not directly k-inductive for any k, but only relative to some auxiliary invariant, so that plain k-induction cannot succeed in proving safety. Strengthening the hypothesis of the inductive-step case with auxiliary invariants may allow the algorithm to prove such properties as well.

Algorithm 3 gets as input initial and maximal values for the loop bound and a function that computes the next loop bound after each iteration (this function can for example increase the value by one, or double it). Additionally we give the algorithm a combination of CPAs (as a composite CPA) that includes the Location CPA $\mathbb{L}$ (cf. Sect. 2.2), our Predicate CPA $\mathbb{P}$ in the

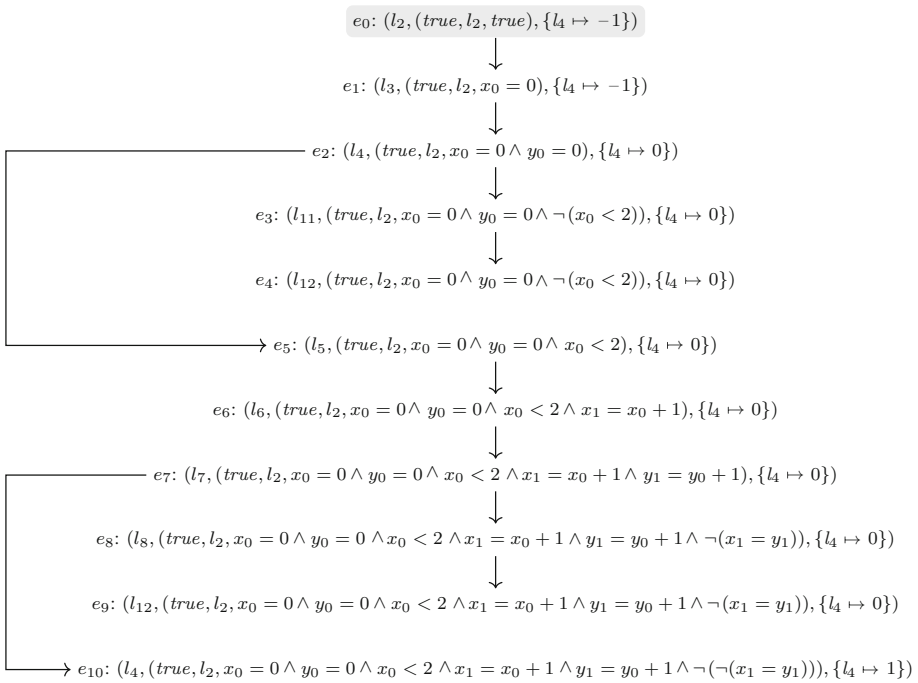


Fig. 5 ARG for applying BMC to the example of Fig. 2

configuration for bounded model checking, and the Loop-Bound CPA $\mathbb{LB}$ (cf. Sect. 4.1). Thus, each abstract state is a tuple of the current program counter l (this is an abstract state of $\mathbb{L}$), a predicate abstract state (which is itself a tuple of an abstraction formula, an abstraction location, and a path formula), and a mapping of loop heads to loop counters (this is an abstract state of $\mathbb{LB}$).

For each value of the loop bound k as determined by the initial and maximal values and the increment function, the algorithm performs the checks for base case, forward condition, and step case. For the *base case* (lines 6–11), which is identical to bounded model checking, we set the bound of the Loop-Bound CPA to k and use the CPA++ algorithm (Algorithm 2) to unroll the program with an abstract state e_{INIT} at the initial program location as initial abstract state and the precision π_{INIT} as initial precision (the Location CPA has an empty precision, the Predicate CPA has a precision that maps all program locations to an empty set of predicates, and the Loop-Bound CPA has a precision that consists of the single constant value k). Then we create a disjunction of the path formulas of all resulting abstract states at the error location. Because of the configuration of the Predicate CPA and the Loop-Bound CPA, this formula represents all paths from l_{INIT} to l_{ERR} that visit the loop body at most k times. If this formula is feasible, l_{ERR} is reachable and the algorithm terminates.

For the *forward condition* (lines 12–14), we check in a similar manner whether the loop-head location l_{LH} is reachable at the start of the $k + 1^{\text{st}}$ loop iteration. If this is not the case, this implies that the error location is also not reachable in the $k + 1^{\text{st}}$ loop iteration (or later on), and thus the program is safe and the algorithm terminates.

For the *inductive-step case* (lines 15–23), we again use the CPA++ algorithm to unroll the program, though this time with a loop bound of $k + 1$ and an abstract state at the loop head

Algorithm 3 Iterative-Deepening k-Induction with Invariants (adapted from [13])

Input:
the initial value $k_{init} \geq 1$ for the bound k,
an upper limit k_{max} for the bound k,
a function $\mathsf{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \mathsf{inc}(n) > n$ for increasing the bound k,
a composite CPA $\mathbb{D}$ with the Location CPA $\mathbb{L}$, the Predicate CPA $\mathbb{P}$, and the Loop-Bound CPA $\mathbb{LB}$ as components,
for which E denotes the set of composite abstract states and Π the set of precisions
Output: **false** if l_{ERR} is reachable, **true** otherwise
Variables: the current loop bound $k \in \mathbb{N}$,
two abstract states $e_{INIT} \in E$ and $e_{LH} \in E$ and a precision $\pi_{INIT} \in \Pi$,
two sets reached and waitlist of elements of $E \times \Pi$, and
a function $\mathsf{abort} : E \to \mathbb{B}$

1: $k := k_{init}$
2: $e_{INIT} := (l_{INIT}, (true, l_{INIT}, true), \{l_{LH} \mapsto -1\})$ // Create abstract state at l_{INIT}.
3: $e_{LH} := (l_{LH}, (true, l_{LH}, true), \{l_{LH} \mapsto 0\})$ // Create abstract state at loop head l_{LH}.
4: $\mathsf{abort}^{never} := \{\cdot \mapsto false\}$ // abort^{never} always returns *false*.
5: **while** $k \leq k_{max}$ **do**
6: $\pi_{INIT} := \{(\emptyset, \{\cdot \mapsto \emptyset\}, k)\}$ // Create initial precision.
7: reached := waitlist := $\{(e_{INIT}, \pi_{INIT})\}$
8: (reached, waitlist) := CPA++($\mathbb{D}$, reached, waitlist, abort^{never})
9: $base\_case := \bigvee \{\varphi \mid ((l_{ERR}, (\cdot, \cdot, \varphi), \cdot), \cdot) \in \mathsf{reached}\}$
10: **if** sat(*base_case*) **then**
11: **return false**
12: $forward\_condition := \bigvee \{\varphi \mid ((l_{LH}, (\cdot, \cdot, \varphi), i), \cdot) \in \mathsf{reached} \wedge i(l_{LH}) = k\}$
13: **if** $\neg$ sat(*forward_condition*) **then**
14: **return true**
15: $\pi_{INIT} := \{(\emptyset, \{\cdot \mapsto \emptyset\}, k+1)\}$ // Initial precision with loop bound $k+1$.
16: reached := waitlist := $\{(e_{LH}, \pi_{INIT})\}$
17: reached := CPA++($\mathbb{D}$, reached, waitlist, abort^{never})
18: $step\_case := \bigvee \{\varphi \mid ((l_{ERR}, (\cdot, \cdot, \varphi), i), \cdot) \in \mathsf{reached} \wedge i(l_{LH}) = k\}$
19: **repeat**
20: *Inv* := get_currently_known_invariant()
21: **if** $\neg$ sat(*Inv* $\wedge$ *step_case*) **then**
22: **return true**
23: **until** *Inv* = get_currently_known_invariant()
24: $k := \mathsf{inc}(k)$
25: **return unknown**

as initial abstract state. For the following satisfiability check, we use the disjunction of the path formulas of all abstract states at the error location and with a loop-counter value of k (i.e., in the $k + 1^{\text{st}}$ loop iteration). Note that because we assume that the loop body cannot be reached from the error location l_{ERR}, this formula represents all paths with k safe loop iterations and a specification violation in the $k + 1^{\text{st}}$ iteration. Additionally, we strengthen the hypothesis of the inductive-step case with the currently known loop invariant that is produced by the concurrently running (external) invariant generator. The invariant obtained from the invariant generator is an SMT formula that is guaranteed to hold at the loop-head location. If the invariant generator produces a stronger loop invariant while the inductive-step case is running, we immediately try again with the new invariant (this can be done efficiently using an incremental SMT solver). If the inductive-step case succeeds, the program is safe and the algorithm terminates. Otherwise, we repeat with a larger value of k, which is called iterative deepening.

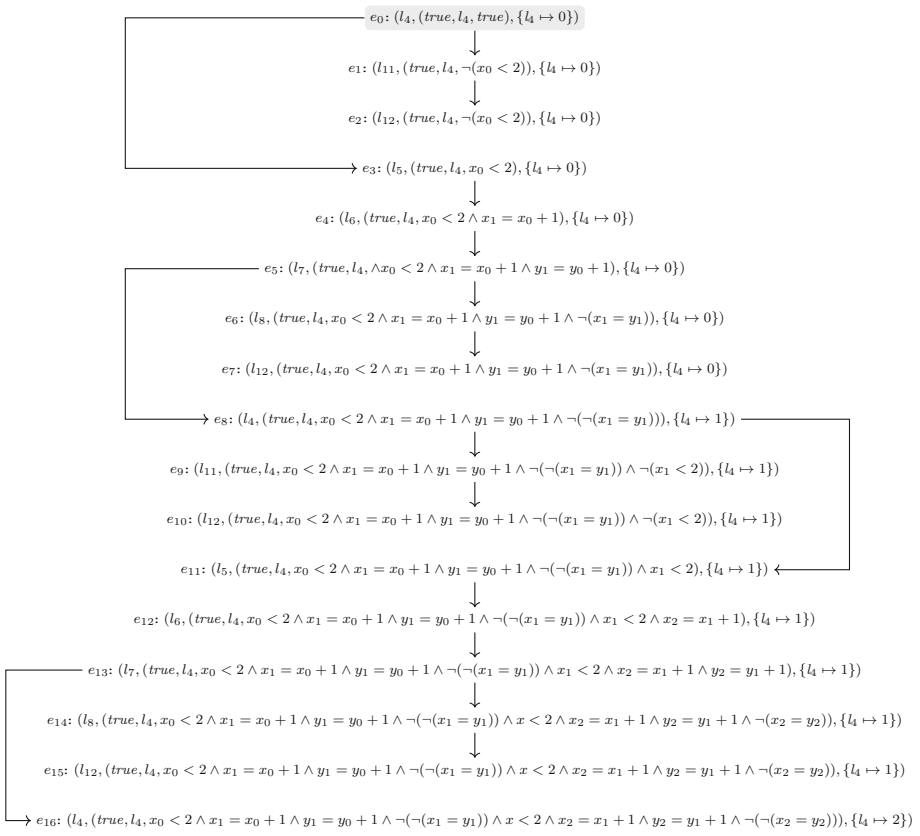


Fig. 6 ARG for the inductive-step case of k-induction applied to the example of Fig. 2

Example 7 (*k-Induction*) If we apply k-induction with $k = 1$ to the program of Fig. 2, the first phase, which is equivalent to BMC, yields the same ARG as in Fig. 5. Figure 6 shows the ARG of the second phase, which is constructed by unrolling the CFA starting at loop head $l_{LH} = l_4$ and using loop bound $k + 1 = 2$. The path formula of the abstract state e_{14} at the error location $l_{ERR} = l_8$, which is in the $k + 1^{\text{st}}$ loop iteration, is unsatisfiable (specifically, the part $\neg(\neg(x_1 = y_1)) \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge \neg(x_2 = y_2)$ is contradictory). This means that after going through one loop iteration without reaching l_8, we can also not reach l_8 in the following loop iteration. In combination with the base case (BMC) from the first phase this proves that the program is safe. Note that this inductive proof is strong enough to prove safety even if we replace the loop condition in line 4 of the example program by a nondeterministic value.

Also note that in this example, no strengthening with auxiliary invariants is required, because the verified property (unreachability of the error location l_8) itself is inductive. Since this is not the case in general, we usually first conjoin auxiliary invariants to the path formula of the abstract state before checking satisfiability. In this example, an auxiliary-invariant generator based on an interval abstract domain might yield the inductive invariant $x >= 0 \wedge x <= 2$, which we would instantiate as $x_1 >= 0 \wedge x_1 <= 2$ for the loop head state of the first iteration.

Algorithm 4 CEGAR($\mathbb{D}$, e_{INIT}, π_{INIT}) for CPAs

Input: a composite CPA $\mathbb{D}$ that is composed of the Location CPA $\mathbb{L}$, the ARG CPA $\mathbb{A}$, and possibly other CPAs,
for which E denotes the set of composite abstract states and Π the set of precisions,
with additional operators fcover and refine,
and an initial abstract state $e_{INIT} = (l_{INIT}, \cdots) \in E$ with initial precision $\pi_{INIT} \in \Pi$
Output: **false** if l_{ERR} is reachable, **true** otherwise
Variables: two sets reached and waitlist of elements of $E \times \Pi$ and
a function abort : $E \to \mathbb{B}$

```
1: reached := {(e_INIT, π_INIT)}
2: waitlist := {(e_INIT, π_INIT)}
3: abort_ERR := {(l, ···) ↦ (l = l_ERR)}          // abort_ERR returns true for abstract error states.
4: loop
5:    (reached, waitlist) := CPA++(𝔻, reached, waitlist, abort_ERR)
6:    if ∃((l_ERR, ···), ·) ∈ reached then
7:       (reached, waitlist) := refine(reached, waitlist)
8:       if ∃((l_ERR, ···), ·) ∈ reached then
9:          return false                          // refine has detected a feasible error path.
10:   else
11:      return true
```

4.3 Lazy Predicate Abstraction

Predicate abstraction with counterexample-guided abstraction refinement (CEGAR) does not use a loop bound, but attempts to converge by determining whether new abstract states are covered by any existing abstract state. In order to make the coverage checks efficient, the abstraction formula of an abstract state overapproximates the reachable concrete states using a boolean combination of predicates over program variables from a given mapping from program locations to sets of predicates (the *precision* π). This abstraction is computed by an SMT solver and the result (the abstraction formula ψ) is stored as a BDD, which can be efficiently checked for entailment. With ABE, the abstraction computations and coverage checks are done only at block ends. For the CPA++ algorithm to terminate it has to be ensured that all ABE blocks do not contain potentially infinite paths, e.g., by using blk^l to let blocks end at loop-head locations. For predicate abstraction we do not use forced coverings.

Furthermore, we wrap our CPA++ algorithm (Algorithm 2) inside Algorithm 4, which implements CEGAR by alternately calling the CPA++ algorithm in order to expand the abstract model and a refinement operator in order to refine the precision of the analysis. We give it a composite CPA that consists of the Location CPA $\mathbb{L}$, the ARG CPA $\mathbb{A}$ (necessary for constructing abstract paths during refinement), and the Predicate CPA $\mathbb{P}$. Using CEGAR and the predicate-refinement strategy of the refinement operator $\mathsf{refine}_{\mathbb{P}}$, it is often possible to find a suitable precision automatically, starting with an empty initial precision. First, CEGAR uses the CPA++ algorithm in order to create the abstract model of the program. If the analysis encounters an abstract state at error location l_{ERR}, we pause the state-space exploration done by CPA++ algorithm (via the function abort_{ERR}) and start the refinement using $\mathsf{refine}_{\mathbb{P}}$. As described in Sect. 3.2, this operator reconstructs the concrete program path leading to the abstract state at l_{ERR} and checks the path for feasibility using an SMT solver. If the concrete error path is feasible, we terminate the analysis. Otherwise, the precision is refined (by employing an SMT solver to compute Craig interpolants [38] for the locations on the error path), and the CPA++ algorithm is restarted with adjusted sets reached and waitlist. Due to the refined precision, it is guaranteed that the previously

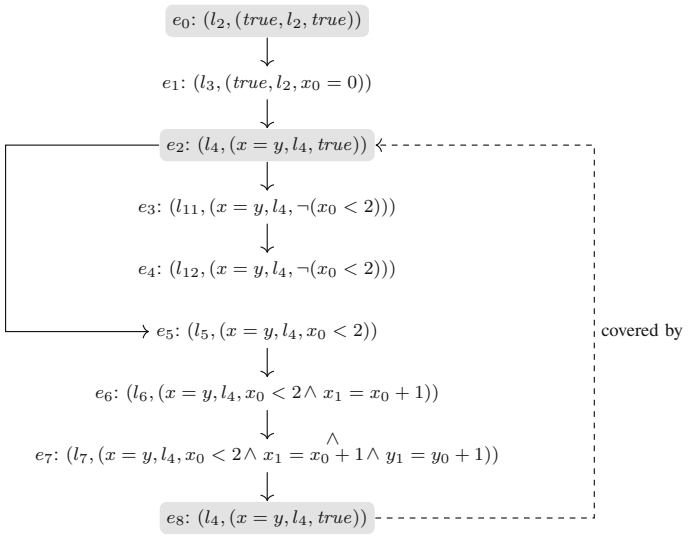


Fig. 7 ARG for predicate abstraction applied to the example of Fig. 2; highlighted nodes are abstraction states

identified infeasible error paths are not encountered again. This process is iterated until either a feasible concrete error path is found, or the CPA++ algorithm terminates proving the program safe.

Example 8 (*Lazy Predicate Abstraction*) If we apply predicate abstraction to the example in Fig. 2 using a precision π with $\pi(l_4) = \{x = y\}$, $\pi(l_8) = \{false\}$, and $\pi(l) = \{\}$ for all other $l \in L$ and defining blocks to end at the loop head l_4 and the error location l_8 (with blk^l), we obtain the ARG depicted in Fig. 7: The first block consists of the abstract states e_0 at location l_2 and e_1 at location l_3. If the analysis hits location l_4, which is a loop head, the path formula $x_0 = 0 \wedge y_0 = 0$ is abstracted using the set of predicates mapped to this location by π. The set of predicates for the location l_4 contains only the predicate $x = y$, which is implied by the path formula and becomes the abstraction formula of the new abstraction state e_2, while the path formula of e_2 is reset to *true*. From that point onward, there are two possible paths: one directly to the end of the program if `x` is greater than or equal to `2`, and another one into the loop if `x` is less than `2`. The path avoiding the loop (abstract states e_3 and e_4) is trivially safe, because from l_{11} or l_{12} there is no control-flow path back to the error location. The path through the loop (abstract states e_5, e_6, and e_7) increments both variables before encountering the assertion. At the error location l_8 the block operator forces an abstraction computation, which in this case is equivalent to a satisfiability check because the precision contains only the predicate *false* for location l_8. Because the combination of the abstraction formula $x = y$ that encodes the reachability of the block entry and the current path formula is unsatisfiable, the error location is not reachable at this point. Thus, the only successor of e_7 is at the loop head l_4, which causes the previous block to end. The abstraction computation yields again the abstraction formula $x = y$ at l_4 (cf. Example 2), which is already covered by the abstract state e_2. Therefore, unrolling the CFA into the ARG completed without encountering the error location $l_{ERR} = l_8$. The algorithm thus concludes that the program is safe.

4.4 Lazy Abstraction with Interpolants (IMPACT)

Lazy abstraction with interpolants [61], more commonly known as the IMPACT algorithm due to its first implementation in the tool IMPACT, was originally presented as an algorithm that repeatedly executes the steps EXPAND (discovery of new abstract states), REFINE (strengthening of abstract states using interpolation), and COVER (detecting coverage between abstract states). Later on it was reformulated in a unified framework together with predicate abstraction and enhanced with ABE [25]. Our description here is based on this reformulation, which was shown to behave similarly to the original algorithm. Like for predicate abstraction, for IMPACT we use CEGAR (Algorithm 4), the CPA++ algorithm, and the Predicate CPA, however, we configure the latter differently. Compared to predicate abstraction, π stays always empty because the IMPACT refinement strategy of $\mathsf{refine}_{\mathbb{P}}$ is used. Thus, the abstraction computation at block ends always trivially returns *true*. The IMPACT refinement strategy, however, makes use of the fact that interpolants are guaranteed to hold at their specific location in the error path and directly strengthens the abstraction formulas of abstract states along the error path with the respective interpolants. The abstract error state is removed during refinement and all coverage relations involving the strengthened abstract states are rechecked after refinement. Furthermore, the abstraction formulas ψ are stored syntactically and coverage is checked using an SMT solver, instead of BDD entailment. If desired, we can configure $\mathsf{fcover}_{\mathbb{P}}$ to perform interpolation-based forced covering as an optimization (cf. Sect. 3.3). IMPACT avoids the costly abstraction computations and rediscovery of abstract states, at the expense of more costly coverage checks.

> *Example 9* (IMPACT) If we apply the IMPACT approach to the example program from Fig. 2, define blocks to end at the loop head l_4 and assume that both interpolations that are required during the analysis yield the interpolant $x = y$ at location l_4, we obtain an ARG as depicted in Fig. 8: Starting with the initialization of the variables, we first obtain the abstract states e_0 and e_1; at e_2, however, we reset the path formula to *true*, because l_4 is a block entry. Note that at this point, the abstraction formula for this block is still *true*. Unwinding the first loop iteration, we first obtain abstract states for incrementing the variables and then hit the error location $l_{ERR} = l_8$ with abstract state e_8. Thus we start a refinement using $\mathsf{refine}_{\mathbb{P}}$ with the IMPACT refinement strategy. An SMT check on the reconstructed concrete error path shows that the path is infeasible, therefore, we perform an interpolation. For the example we assume that interpolation provides the interpolant $x = y$, so we strengthen the abstraction formula of e_2 with this interpolant and strengthen the abstraction formula of e_8 with *false* (cf. Example 4). Because e_8 now represents an empty set of concrete states, we remove it from the ARG. Then, we continue the expansion of e_7 towards l_4 with abstract state e_9. Note that at this point, the abstraction formula for e_9 is still *true*, thus e_9 is not covered by e_2 with $x = y$. Also, e_2 cannot be covered by e_9, because e_2 is an ancestor of e_9. We unwind the loop for another iteration and again hit the error location l_8 with abstract state e_{13}. Once again, the concrete path formula for this abstract state is infeasible, so we interpolate. For the example we assume that interpolation provides again the interpolant $x = y$, and use it to strengthen the abstraction formula of e_9. The abstract error state e_{13} is removed from the ARG after its abstraction formula is strengthened to *false*. Now, a coverage check reveals that e_9 is covered by e_2, because neither e_9 nor any of its ancestors is covered yet, both belong to the same location l_4, $x = y$ implies $x = y$, e_9 is not an ancestor of e_2, and e_2 was created before e_9. Because e_9 is now covered, we need not continue expanding any of its (transitive) successors, and the algorithm terminates without finding any feasible error paths, thus proving safety.

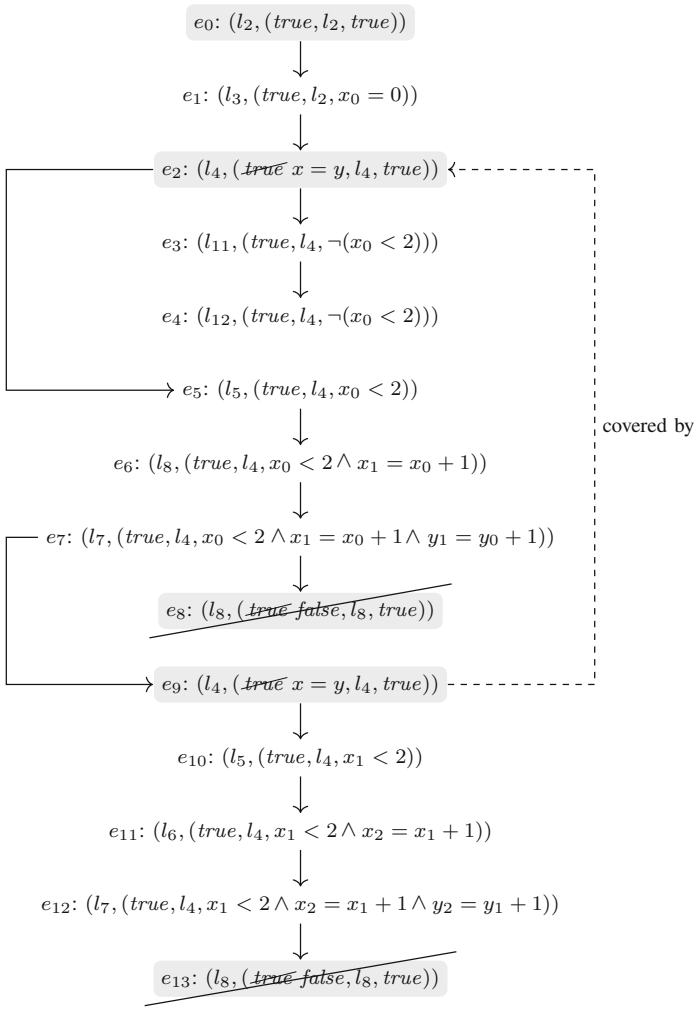


Fig. 8 Final ARG for applying the IMPACT approach to the example of Fig. 2; highlighted nodes are abstraction states

4.5 Summary

We showed how to express four approaches to software verification with our framework for predicate-based analyses and illustrated how they work on the example from Fig. 2. Table 1 summarizes the choices that need to be made for each of the approaches. While BMC is limited in its capacity of proving correctness, it is also the most straightforward of the four approaches, because k-induction requires an auxiliary-invariant generator to be applicable in practice, and predicate abstraction and IMPACT require interpolation techniques. While the invariant generator and the interpolation engine are usually treated as black box in the description of these approaches, the efficiency and effectiveness of the techniques depends on the quality of these modules.

Table 1 Configuration of the predicate CPA $\mathbb{P}$ for the four approaches

	Abstraction-formula representation	blk	Refinement strategy	$\mathsf{fcover}_{\mathbb{P}}$
BMC	SMT	blk^{never}	None	fcover^{id}
k-Induction	SMT	blk^{never}	None	fcover^{id}
Predicate abstraction	BDD	e.g. blk^{l}	Predicate refinement	fcover^{id}
IMPACT	SMT	e.g. blk^{l}	IMPACT refinement	e.g. $\mathsf{fcover}^{\text{IMPACT}}$

4.5.1 Further Algorithms

There are other approaches for software verification besides the four that we unify in this work, and of course, the best features of all approaches can be combined into new, "hybrid" methods, such as implemented in CPACHECKER [71], SEAHORN [48], and UFO [3]. The focus of this article is not to find the best possible combination, but to study the approaches in isolation. In the following, we briefly discuss the most important SMT-based approaches, ordered roughly accordingly to how similar they are to the approaches that we have discussed so far.

The UFO algorithm [2] combines the IMPACT algorithm with predicate abstraction. UFO is similar to IMPACT, but implements a choice between performing predicate-abstraction computation when creating fresh abstract states and initializing them with *true* as IMPACT does. Refinement is done using interpolation, and the interpolants can be used to either strengthen the abstract states (pure IMPACT behavior), or to update the set of predicates (pure predicate-abstraction behavior), or do both. This approach can be seen as an instantiation of our framework with a refinement operator that uses both the IMPACT- and the predicate-refinement strategies (cf. Sect. 3.2).

Symbolic execution [58] follows each path in the program separately and interprets its operations; the abstract states track explicit and symbolic values of program variables in a symbolic store as well as constraints over the symbolic values. If a variable is assigned a nondeterministic value, a fresh symbolic value is stored; if an explicit value can be determined by the analysis, then the explicit value is stored. Constraints that are encountered along a path are tracked and checked for satisfiability, using the symbolic store as interpretation, whenever the feasibility of the path needs to be determined (e.g., if an error location is reached). The framework presented in this work can be configured as an analysis that behaves similarly to symbolic execution (just without symbolic store) by using the CPA algorithm with the Predicate CPA configured to use blk^{never} and merge^{sep} instead of $\mathsf{merge}_{\mathbb{P}}$. The operator blk^{never} has the effect of disabling abstraction computations and thus accumulating the semantics of all program operations of a path in the path formula of abstract states during traversal (as for BMC). The operator merge^{sep} has the effect of preventing all merges between abstract states and thus keeping all paths separate, forming a reachability tree. Note that differently from symbolic execution this configuration tracks all values syntactically.

Slicing abstractions [30,43] (a.k.a. "state splitting") starts with an abstract-reachability graph in which all abstract states are labeled with *true*. The algorithm iteratively searches for an infeasible error path in this graph and computes interpolants for the respective path. The strategy for refining the abstract model consists of duplicating each abstract state for which an interpolant was found (including its edges) and conjoining the interpolant to one of the resulting abstract states and the negated interpolant to the other one ("state splitting").

Then all edges of both resulting states are checked for feasibility. This always results in enough edges being removed such that the current infeasible error path no longer exists in the abstract-reachability graph. This is repeated (CEGAR) until either no infeasible error path exists anymore, or a feasible error path is found. The approach of splitting abstract states has also been extended to a combination of predicate abstraction and explicit-value analysis [49], similar to the combination of lazy predicate abstraction and explicit-value analysis [22].

Trace abstraction [50] is a CEGAR-based approach in which the iteratively refined abstract model of the program is not a set of abstract states, but instead an automaton that represents an overapproximation of the feasible paths of the program. Every time a spurious counterexample is detected, a trace automaton that represents a set of infeasible paths including the current counterexample is created using interpolation, and this trace automaton is subtracted from the current abstract model.

Software proof-based abstraction with counterexample-based refinement (SPACER) [59] is an approach that combines CEGAR with its dual, proof-based abstraction (PBA) [62]. While CEGAR maintains an overapproximation of the program and refines it using infeasible error paths, PBA maintains an underapproximation and refines it if it finds a safety proof that holds only for the underapproximation but not for the original system. SPACER follows the PBA approach but uses an abstraction of the underapproximation to allow handling infinite-state systems and refines this abstraction using CEGAR.

Model checking modulo theories (MCMT) [45,46] is an approach that focuses on verifying infinite-state systems that use arrays. It is based on a backwards-reachability analysis and SMT solving for theories that fulfill certain conditions. MCMT has been combined with CEGAR and interpolation to define an analysis that can be described as a backwards variant of IMPACT and applied to software model checking [4]. This approach uses interpolation to compute quantifier-free interpolants for a restricted class of formulas with arrays and can prove universally quantified properties over arrays automatically.

IC3 [28], which is also known as property-directed reachability (PDR) [42], is an algorithm for model checking finite-state systems. It aims at producing an inductive invariant that is strong enough to prove safety by incrementally learning clauses that are inductive with regard to the previously learned clauses. Such clauses are derived by generalizing from counterexamples to induction proofs. PDR was originally designed for boolean transition systems and based on SAT solving. It has been generalized from boolean systems to SMT [52] and applied to software in various ways [27,32,33,54], which we discuss in the following. If PDR is combined with an explicit (instead of symbolic) tracking of the program counter, this lets the algorithm produce an abstract-reachability tree [32]. In fact, because the sets of clauses that PDR learns fulfill the properties of interpolants, this tree-based PDR can even be seen as a version of IMPACT, just with a different way of producing interpolants. A hybrid approach that uses both a regular interpolation engine as well as PDR for producing interpolants is also possible [32]. It would be an interesting extension of our Predicate CPA to adopt the clause-learning strategy of PDR as an alternative to using interpolation during refinement (cf. Sect. 3.2). Another approach for software verification using PDR is to define a boolean abstract model of the program using predicate abstraction and use an almost unchanged PDR algorithm for verifying the abstract model [33]. The abstraction is refined using typical predicate-discovery strategies (e.g., interpolation) whenever an infeasible error path is found. CTIGAR [27] is an approach for applying PDR to software that does not rely on CEGAR (i.e., using error paths for refinement), but uses counterexamples to induction (CTI) for abstraction refinement. CTIGAR computes abstract CTIs from the concrete CTIs of PDR by using predicate abstraction and refines the abstraction using interpolation if it finds a clause that is inductive with regard to the previously learned clauses, but its abstract version is not.

PDR can also be extended from standard induction to property-directed k-induction [54]. This allows it to more easily verify programs for which useful 1-inductive invariants are cumbersome and difficult to find, while more concise k-inductive invariants exist.

Loop invariants that are strong enough to verify program safety can also be computed via abduction [40]. Similar to the PDR-based approaches, a candidate invariant is strengthened until it becomes inductive. However, while PDR starts from facts that are known to hold, the abductive approach starts from the conjecture it wants to prove and asks an abduction engine to generate candidate strengthenings that would allow the conjecture to hold. Then it needs to check whether one of the candidates holds, which may need further recursive strengthenings with backtracking. As abduction engine, it is possible to use for example quantifier elimination in Presburger arithmetic.

5 Evaluation

We evaluate BMC, k-induction, predicate abstraction, and IMPACT on a large set of verification tasks and compare the approaches.

5.1 Benchmark Set

As benchmark set we use the verification tasks from the 2017 Competition on Software Verification (SV-COMP'17) [10]. We used only verification tasks where the property to verify is the reachability of a program location (excluding the properties for memory safety, overflows, and termination, which are not in our scope). From the remaining set of verification tasks, we excluded the categories *ReachSafety-Arrays*, *ReachSafety-Floats*, *ReachSafety-Recursive*, and *ConcurrencySafety*, each of which is not supported by at least one of our implementations of the approaches. The resulting set of categories consists of a total of 5287 verification tasks from the subcategory *DeviceDriversLinux64_ReachSafety* of the category *SoftwareSystems* and from the following subcategories of the category *ReachSafety*: *Bitvectors*, *ControlFlow*, *ECA*, *Floats*, *Heap*, *Loops*, *ProductLines*, and *Sequentialized*. A total of 1374 tasks in the benchmark set contain a known specification violation, while the rest of the tasks is assumed to be free of violations.

5.2 Experimental Setup

Our experiments were conducted on machines with one 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM each. The operating system was Ubuntu 16.04 (64 bit), using Linux 4.4 and OpenJDK 1.8. Each verification task was limited to two CPU cores, a CPU run time of 15 min, and a memory usage of 15 GB. We used the benchmarking framework BENCHEXEC[3] [23] to perform our experiments. We used version `1.6.18-jar17` of CPACHECKER, with MATHSAT5 as solver for all SMT queries. We configured CPACHECKER to use the SMT theories of equality with uninterpreted functions, bit vectors, and floats. For IMPACT and predicate abstraction, an ABE block size needs to be chosen: we used blk^l to let blocks end at loop heads. For IMPACT we also activated the forced-covering optimization with $\mathsf{fcover}^{\text{IMPACT}}$. For BMC we used a configuration with forward-condition checking [44]. For BMC and k-induction, we used an initial bound of $k = 1$ and an increment function $inc(n) = n + 1$. Auxiliary invariants are provided to k-induction using a continuously refining data-flow analysis from existing work [14] that

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

uses disjunctions of intervals as its abstract domain. We configure CPACHECKER to avoid false alarms by validating the feasibility of each found error path using CBMC 5.6. Time results are rounded to two significant digits.

5.3 Reproducibility

All presented approaches are implemented in the open-source verification framework CPACHECKER [20], which is available under the Apache 2.0 license. All experiments are based on publicly available benchmark verification tasks [10]. Tables with our detailed experimental results are available on the supplementary web page.[4]

5.4 Experimental Validity

5.4.1 Internal Validity

We implemented all evaluated approaches using the same software-verification framework: CPACHECKER. This allows us to compare the actual algorithms instead of comparing different tools with different front ends and different utilities, thus eliminating influences on the results caused by implementation differences that are unrelated to the actual algorithms.

To ensure technical accuracy, we used the open-source benchmarking framework BENCHEXEC[5] [23] for conducting our experiments.

5.4.2 External Validity

We perform our experiments on the largest, most diverse, and publicly available collection of verification tasks,[6] which is also used by the international competition on software verification.

5.5 Results Overall

Table 2 shows the number of correctly solved verification tasks for each of the approaches, as well as the time that was spent on producing these results. None of the approaches reported incorrect proofs[7] or incorrect alarms. When an algorithm exceeds its time or memory limit, it is terminated inconclusively. Other inconclusive results occur, for example, if the implementation encounters an unsupported feature, such as recursion, or if during an SMT query, an error occurs in the SMT solver. When comparing k-induction to the other techniques, there is sometimes a chance that the other techniques must give up due to an unsupported feature, while k-induction is not encountering the unsupported feature because it is waiting for the invariant generator to generate a strong invariant. Therefore, k-induction has fewer other inconclusive results but instead more timeouts than predicate abstraction and IMPACT. The quantile plots in Fig. 9 show the accumulated number of successfully solved verification tasks within a given amount of CPU time. A data point (x, y) of a graph means that for the respective configuration, x is the number of correctly solved tasks with a CPU run time of less than or equal to y seconds.

[4] https://www.sosy-lab.org/research/k-ind-compare

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

[6] https://github.com/sosy-lab/sv-benchmarks

[7] For BMC, real proofs are accomplished by successful forward-condition checks, which prove that no further unrolling is required to exhaustively explore the state space.

Table 2 Experimental results of the approaches for all 5287 verification tasks, 1374 of which contain bugs, while the other 3913 are considered to be safe

Algorithm	BMC	k-Induction	Predicate abstraction	IMPACT
Correct results	1043	**2600**	2506	2499
Correct proofs	666	**2237**	2169	2143
Correct alarms	**377**	363	337	356
Timeouts	3365	2375	2099	2442
Out of memory	603	232	78	139
Other inconclusive	276	80	604	207
Times for correct results				
Total CPU time (h)	5.7	34	28	27
Avg. CPU time (s)	20	47	40	39
Total wall time (h)	4.9	17	24	24
Avg. wall time (s)	17	24	34	34
Times for correct proofs				
Total CPU time (h)	2.9	28	23	23
Avg. CPU time (s)	16	45	37	39
Total wall time (h)	2.4	14	19	20
Avg. wall time (s)	13	23	32	34
Times for correct alarms				
Total CPU time (h)	2.8	6.2	5.4	4.1
Avg. CPU time (s)	27	61	57	41
Total wall time (h)	2.5	3.2	4.9	3.7
Avg. wall time (s)	24	32	52	38

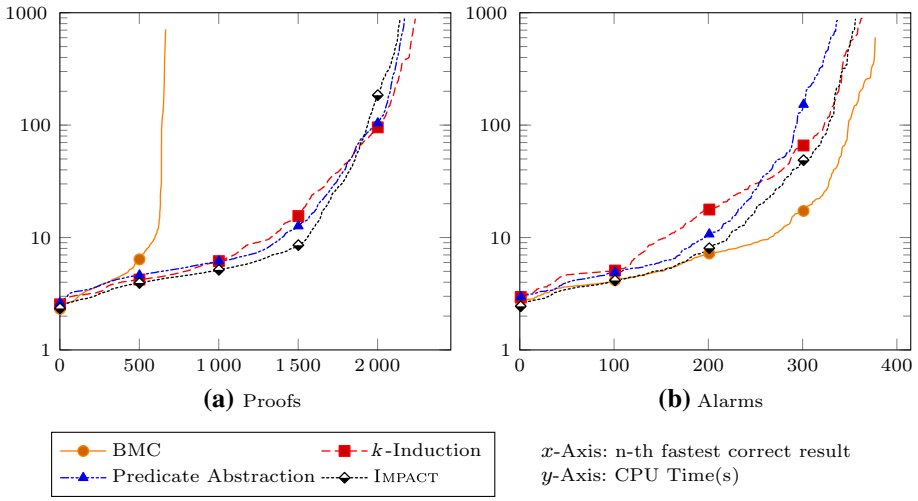


Fig. 9 Quantile plots for all correct proofs and alarms

5.5.1 BMC

As expected, BMC produces both the fewest correct proofs and the most correct alarms, confirming BMC's reputation as a technique that is well suited for finding bugs. Having the fewest solved tasks, BMC also accumulates the lowest total CPU time for correct results. Its average CPU time spent on correct results is also lower than for the other techniques: for proofs, BMC often fails to provide a correct result while the other approaches spend a lot of time on successfully finding a proof; for finding bugs, its straightforward approach outperforms the abstraction techniques while k-induction unnecessarily invests time in generating auxiliary invariants. On average, BMC spends 1.2 s on formula creation, 3.5 s on SMT-checking the forward condition, and 7.4 s on SMT-checking the feasibility of error paths.

5.5.2 k-Induction

The slowest technique is k-induction with continuously refined invariant generation, which is the only technique that effectively uses both available cores by running the auxiliary-invariant generation in parallel to the k-induction procedure, thus spending significantly more CPU time than the other techniques, while the wall time it spends is comparable to the wall time spent by the abstraction techniques for correct proofs. Compared to BMC, k-induction spends additional time on building the step-case formula and generating auxiliary invariants, but can often prove safety by induction without unrolling loops. Considering that over the whole benchmark set, k-induction generates the highest overall number of correct results, the additional effort appears to be mostly well spent. On average, k-induction spends 1.2 s on formula creation in the base case, 2.5 s on SMT-checking the forward condition, 3.0 s on SMT-checking the feasibility of error paths, 9.3 s on creating the step-case formula, 14 s on SMT-checking inductivity, and 20 s on generating auxiliary invariants, which shows that the inductive-step case requires much more effort than the base case and also about 3 s more than for invariant generation. For tasks containing actual bugs, however, this effort is wasted, which explains why k-induction spends not only more CPU time but also significantly more wall time on correct alarms than the other techniques.

5.5.3 Predicate Abstraction and IMPACT

Predicate abstraction and IMPACT both perform similarly for finding proofs, which matches the observations from earlier work [25]. An interesting difference is that IMPACT finds more bugs. We attribute this observation to the fact that abstraction in IMPACT is lazier than with predicate abstraction, which allows IMPACT to explore larger parts of the state space in a shorter amount of time than predicate abstraction, causing IMPACT to find bugs sooner. For verification tasks without specification violations, however, the more eager predicate-abstraction technique pays off, because it avoids many SMT-checks for determining coverage. Although in total, both abstraction techniques have to spend similar effort, this effort is distributed differently across the various steps: While, on average, predicate abstraction spends more time on computing abstractions (23 s) than the IMPACT algorithm spends on deriving its abstraction by interpolation (9.0 s), the latter requires the relatively expensive forced-covering step (12 s).

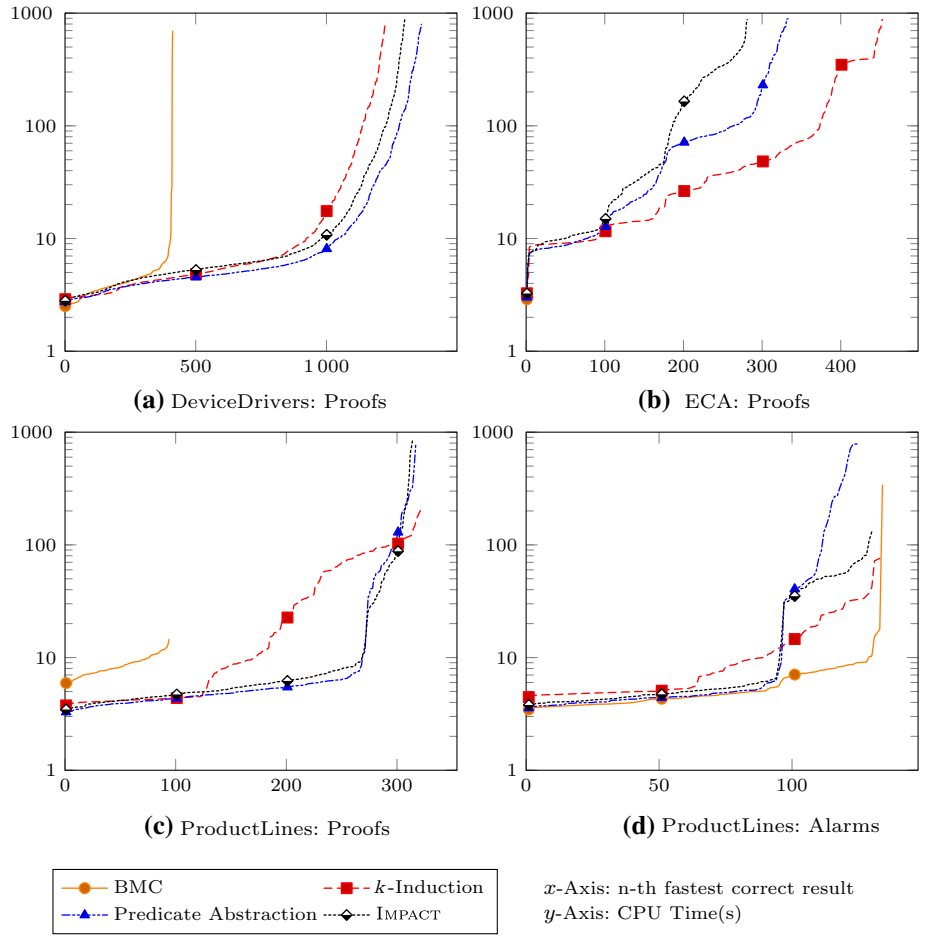


Fig. 10 Quantile plots for some of the categories

5.6 Results on Selected Categories

Although the plot in Fig. 9a suggests that k-induction with continuously refined invariants outperforms the other techniques in general for finding proofs, a closer look at the results in individual SV-COMP categories reveals that the performance of an algorithm strongly depends on the type of verification task, but also reconfirms the observation of Fig. 9b that BMC consistently performs well for finding bugs.

For example, on the safe tasks of the category on Linux device drivers, k-induction performs worse than predicate abstraction and IMPACT (Fig. 10a). These device drivers are often large in size, containing pointer arithmetic and complex data structures. The interval-based auxiliary-invariant generator that we used for k-induction is not a good fit for this kind of problems, and a lot of effort is wasted, while the abstraction techniques are often able to quickly determine that many operations on pointers and complex data structures are irrelevant for the safety property. We did not include the plot for the correct alarms in the category

on device drivers, because each of the approaches only solves about 30 tasks, i.e., there is not enough data among the correct alarms to draw any further conclusions.

The quantile plot for the correct proofs in the category of event condition action systems (ECA) is displayed in Fig. 10b. BMC is hardly visible in this figure, because there is only a single task in the category that it could unroll exhaustively. Each of these tasks only consists of a single loop, but these loops contain complex branching structures over many different integer variables, which leads to an exponential explosion of paths, such that checking satisfiability of an SMT formula representing an unwinding of such a loop is often expensive in terms of time and memory. Also, because in many tasks of this category almost all of the variables are in some way relevant to the reachability of the error location within this complex branching structure, the abstraction techniques are unable to come up with useful abstractions and perform poorly. The interval-based auxiliary-invariant generator that we use for *k*-induction, however, appears to provide useful invariants for handling the complexity of the control structures, and the state-machine-like nature of these tasks requires the consideration of many different cases and their interaction across consecutive loop iterations, such that *k*-induction performs much better than all other techniques in this category. We did not include the plot for the correct alarms in this category, because the abstraction techniques were not able to detect a single bug, and only BMC and *k*-induction detect one single bug for the same task, namely `Problem10_label46_false-unreach-call.c`.

Figure 10c shows the quantile plot for correct proofs in the category on product lines. In this category, as in Fig. 9a, *k*-induction slightly outperforms the other techniques in the number of found proofs but it also becomes even more apparent than in other categories how much slower than the other techniques it is (on average for correct results). Figure 10d shows the quantile plot for correct alarms in the same category. It is interesting to observe that IMPACT distinctly outperforms predicate abstraction on the tasks that require over 40 s of CPU time, whereas in the previous plots, the differences between the two abstraction techniques were either hardly visible or IMPACT performed worse than predicate abstraction. While, as shown in Fig. 10c, both techniques report almost the same amount of correct proofs (317 for predicate abstraction, 315 for IMPACT), IMPACT detects 130 bugs, whereas predicate abstraction detects only 125. This seems to indicate that the state space spanned by the different product-line features can be explored more quickly by lazy abstraction of IMPACT than with the more eager predicate abstraction.

5.7 Results on Selected Verification Tasks Showing Individual Strengths

The previous discussion showed that while overall, the approaches perform rather similar (apart from BMC being inappropriate for finding proofs, which is expected), each of them has some strengths due to which it outperforms the other approaches on certain programs. In the following, we will list some examples from various categories of SV-COMP that were each solved by one of the approaches, but not by the others, and give a short explanation of the reasons.

5.7.1 BMC

Only BMC finds a bug in task `const_false-unreach-call1.i` (23 s, Category *Loops*), and only BMC proves, by exhaustively unrolling a loop, safety for the task `pals_opt-floodmax.4_true-unreach-call.ufo.BOUNDED-8.pals_true-termination.c` (310 s, Category *Sequentialized*). Both of these tasks have in common that they contain bounded loops. The bounded loops are a good fit for BMC and enable it

to prove correctness; k-induction, which in theory is at least as powerful as BMC, spends too much time trying to generate auxiliary invariants and exceeds the CPU time limit before solving these tasks.

5.7.2 k-Induction

k-Induction outperforms the other techniques on many of the state-machine-like tasks of the category on event condition action systems (*ECA*). Only k-induction proves correctness of the task `Problem14_label00_true-unreach-call.c` (14 s, Category *ECA*), which, like all tasks in that category, encodes a complex state machine, i.e., a loop over switch statements with many cases, which in turn modify the variable that is considered by the switch statement. The loop is unbounded, such that BMC cannot exhaustively unroll it, and the loop invariants that are required to prove correctness of the task need to consider the different cases and their interaction across consecutive loop iterations, which is beyond the scope of the abstraction techniques but easy for k-induction (cf. [13] for a detailed discussion of a similar example).

5.7.3 Predicate Abstraction

Only predicate abstraction solves verification task `toy_true-unreach-call_false-termination.cil.c` (65 s, Category *Sequentialized*). The task consists of an unbounded loop that contains a complex branching structure over integer variables, most of which only ever take the values `0`, `1` or `2`. Interpolation quickly discovers the abstraction predicates over these variables that are required to solve the task, but in this example, predicate abstraction profits from eagerly computing a sufficiently precise abstraction early after only 10 refinements while the lazy refinement technique used by IMPACT exceeds the time limit after 165 refinements, and the invariant generator used by k-induction fails to find the required auxiliary invariants before reaching the time limit.

5.7.4 IMPACT

Only IMPACT solves `Problem05_label50_true-unreach-call.c` (190 s, Category *ECA*). BMC fails on this task due to the unbounded loop, and the invariant generator used by k-induction does not come up with any meaningful auxiliary invariant before exceeding the time limit. Predicate abstraction exceeds the time limit after only four refinements, and up to that point, 90% of its time is spent on eagerly computing abstractions. The lazy abstraction performed by IMPACT, however, allows it to progress quickly, and the algorithm finishes after 9 refinements.

6 Conclusion

This paper presents a comparative study of four state-of-the-art approaches for SMT-based software verification. First, we define a configurable program analysis for the predicates domain, which serves as the unifying core component of our comparison framework. Second, we express each approach in our framework by a specific set of parameters and illustrate the effect on how the state-space exploration is performed. Third, we provide the results of a thorough experimental study on a large number of verification tasks, in order to show the effect and performance of the different approaches, including a detailed discussion of particular

verification tasks that can be solved by one approach while all others fail. In conclusion, there is no clear winner: there are disadvantages and advantages for each approach. We hope that our conceptual and experimental overview is useful and contributes to understanding the difference of the approaches and the potential application areas.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA (1986)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: Proceedings of TACAS, LNCS 7214, pp. 157–172. Springer (2012)
3. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: Proceedings of CAV, LNCS 7358, pp. 672–678. Springer (2012)
4. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: An extension of lazy abstraction with interpolation for programs with arrays. Form. Methods Syst. Des. **45**(1), 63–109 (2014)
5. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: technology transfer of formal methods inside microsoft. In: Proceedings of IFM, LNCS 2999, pp. 1–20. Springer (2004)
6. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011)
7. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. In: Proceedings of TACAS, LNCS 2031, pp. 268–283. Springer (2001)
8. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings of POPL, pp. 1–3. ACM (2002)
9. Beckert, B., Hähnle, R.: Reasoning and verification: state of the art and current trends. IEEE Intell. Syst. **29**(1), 20–29 (2014)
10. Beyer, D.: Software verification with validation of results (report on SV-COMP 2017). In: Proceedings of TACAS, LNCS 10206, pp. 331–349. Springer (2017)
11. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proceedings of FMCAD, pp. 25–32. IEEE (2009)
12. Beyer, D., Dangl, M.: SMT-based software model checking: an experimental comparison of four algorithms. In: Proceedings of VSTTE, LNCS 9971, pp. 181–198. Springer (2016)
13. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proceedings of CAV, LNCS 9206, pp. 622–640. Springer (2015)
14. Beyer, D., Dangl, M., Wendler, P.: Combining k-induction with continuously-refined invariants. Technical Report MIP-1503, University of Passau (January 2015). arXiv:1502.00096
15. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. **9**(5–6), 505–525 (2007)
16. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Proceedings of VMCAI, LNCS 4349, pp. 378–394. Springer (2007)
17. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Proceedings of PLDI, pp. 300–309. ACM (2007)
18. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Proceedings of CAV, LNCS 4590, pp. 504–518. Springer (2007)
19. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proceedings of ASE, pp. 29–38. IEEE (2008)
20. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: Proceedings of CAV, LNCS 6806, pp. 184–190. Springer (2011)
21. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proceedings of FMCAD, pp. 189–197. FMCAD (2010)
22. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proceedings of FASE, LNCS 7793, pp. 146–162. Springer (2013)
23. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proceedings of SPIN, LNCS 9232, pp. 160–178. Springer (2015)
24. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proceedings of ISoLA, LNCS 7610, pp. 1–6. Springer (2012)

25. Beyer, D., Wendler, P.: Algorithms for software model checking: predicate abstraction vs. IMPACT. In: Proceedings of FMCAD, pp. 106–113. FMCAD (2012)
26. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of TACAS, LNCS 1579, pp. 193–207. Springer (1999)
27. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Proceedings of CAV, LNCS 8559, pp. 831–848. Springer (2014)
28. Bradley, A.R.: SAT-based model checking without unrolling. In: Proceedings of VMCAI, LNCS 6538, pp. 70–87. Springer (2011)
29. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proceedings of SAS, LNCS 9291, pp. 145–161. Springer (2015)
30. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. Fundam. Inform. **89**(4), 369–392 (2008)
31. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
32. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Proceedings of CAV, LNCS 7358, pp. 277–293. Springer (2012)
33. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Proceedings of TACAS, LNCS 8413, pp. 46–61. Springer (2014)
34. 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)
35. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proceedings of TACAS, LNCS 2988, pp. 168–176. Springer (2004)
36. Colón, M., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Proceedings of CAV, LNCS 2725, pp. 420–432. Springer (2003)
37. Cordeiro, L.C., Morse, J., Nicole, D., Fischer, B.: Context-bounded model checking with ESBMC 1.17 (competition contribution). In: Proceedings of TACAS, LNCS 7214, pp. 534–537. Springer (2012)
38. Craig, W.: Linear reasoning. A new form of the Herbrand–Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)
39. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
40. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: Proceedings of OOPSLA, pp. 443–456. ACM (2013)
41. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Proceedings of SAS, LNCS 6887, pp. 351–368. Springer (2011)
42. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proceedings of FMCAD, pp. 125–134. FMCAD Inc. (2011)
43. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Proceedings of VMCAI, LNCS 7148, pp. 186–201. Springer (2012)
44. Gadelha, M.Y.R., 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 (2017)
45. Ghilardi, S., Ranise, S.: Goal-directed invariant synthesis for model checking modulo theories. In: Proceedings of TABLEAUX, LNCS 5607, pp. 173–188. Springer (2009)
46. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. Log. Methods Comput. Sci. **6**(4) (2010)
47. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Proceedings of CAV, LNCS 1254, pp. 72–83. Springer (1997)
48. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: a framework for verifying C programs (competition contribution). In: Proceedings of TACAS, LNCS 9035, pp. 447–450. Springer (2015)
49. Hajdu, Á., Tóth, T., Vörös, A., Majzik, I.: A configurable CEGAR framework with interpolation-based refinements. In: Proceedings of FORTE, LNCS 9688, pp. 158–174. Springer (2016)
50. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proceedings of SAS, LNCS 5673, pp. 69–85. Springer (2009)
51. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of POPL, pp. 58–70. ACM (2002)
52. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Proceedings of SAT, LNCS 7317, pp. 157–171. Springer (2012)
53. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
54. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Proceedings of FMCAD, pp. 85–92. IEEE (2016)

55. Kahsai, T., Tinelli, C.: PKIND: a parallel k-induction based model checker. In: Proceedings of International Workshop on Parallel and Distributed Methods in Verification, EPTCS 72, pp. 55–62 (2011)
56. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proceedings of Ershov Memorial Conference, LNCS 5947, pp. 165–176. Springer (2009)
57. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of POPL, pp. 194–206. ACM (1973)
58. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
59. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Proceedings of CAV, LNCS 8044, pp. 846–862. Springer (2013)
60. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proceedings of CAV, LNCS 2725, pp. 1–13. Springer (2003)
61. McMillan, K.L.: Lazy abstraction with interpolants. In: Proceedings of CAV, LNCS 4144, pp. 123–136. Springer (2006)
62. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: Proceedings of TACAS, LNCS 2619, pp. 2–17. Springer (2003)
63. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Berlin (1999)
64. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Proceedings of CAV, LNCS 8559, pp. 106–113. Springer (2014)
65. Rocha, H., Ismail, H.I., Cordeiro, L.C., Barreto, R.S.: Model checking embedded C software using k-induction and invariants. In: Proceedings of SBESC, pp. 90–95. IEEE (2015)
66. Schrammel, P., Kroening, D.: 2LS for program analysis (competition contribution). In: Proceedings of TACAS, LNCS 9636, pp. 905–907. Springer (2016)
67. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electron. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006)
68. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proceedings of FMCAD, LNCS 1954, pp. 127–144. Springer (2000)
69. Sinz, C., Merz, F., Falke, S.: LLBMC: A bounded model checker for LLVM's intermediate representation (competition contribution). In: Proceedings of TACAS, LNCS 7214, pp. 542–544. Springer (2012)
70. Wahl, T.: The k-induction principle. Available at http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf (2013)
71. Wendler, P.: CPACHECKER with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proceedings of TACAS, LNCS 7795, pp. 613–615. Springer (2013)

Software Verification with PDR: An Implementation of the State of the Art

Dirk Beyer[1] and Matthias Dangl[1]

LMU Munich, Germany

Abstract. Property-directed reachability (PDR) is a SAT/SMT-based reachability algorithm that incrementally constructs inductive invariants. After it was successfully applied to hardware model checking, several adaptations to software model checking have been proposed. We contribute a replicable and thorough comparative evaluation of the state of the art: We (1) implemented a standalone PDR algorithm and, as improvement, a PDR-based auxiliary-invariant generator for k-induction, and (2) performed an experimental study on the largest publicly available benchmark set of C verification tasks, in which we explore the effectiveness and efficiency of software verification with PDR. The main contribution of our work is to establish a reproducible baseline for ongoing research in the area by providing a well-engineered reference implementation and an experimental evaluation of the existing techniques.

Keywords: Software verification · Program analysis · Invariant generation · Property-directed reachability (PDR) · IC3 · k-Induction· VVT · CPAchecker

1 Introduction

Automatic software verification [24] is a broad research area with many success stories and large impact on technology that is applied in industry [2, 14, 27]. It complements other general approaches to ensure functional correctness, like software testing [31] and interactive software verification [3]. One large sub-area of automatic software verification includes algorithms and approaches that are based on SMT technology. Classic approaches like bounded model checking [10], predicate abstraction [1, 19], and k-induction [5, 26, 32] are well understood and evaluated; a recent survey [6] provides a uniform overview and sheds light on the differences of the algorithms. Property-directed reachability (PDR) [12] is a relatively recent (2011) approach that is not yet included in comparative evaluations that go beyond applying different implementations of the same or different techniques to a set of benchmark tasks, but additionally pair such experiments with a discussion of how the concepts can be expressed in a common formalism. The approach was originally applied to transition systems from hardware designs, but was also adapted to software verification [11, 12, 13, 15, 16, 25, 28, 29].

An extended version of this article is available as technical report [8].
A replication package is available on Zenodo [9].

A. Biere and D. Parker (Eds.): TACAS 2020, LNCS 12078, pp. 3–21, 2020.
https://doi.org/10.1007/978-3-030-45190-5_1

While in theory, given the aforementioned body of work on the topic, the advantages and disadvantages of using PDR seem clear, we are interested in understanding the effect of applying PDR to a large set of verification tasks that were collected from academia and also from industrial software, such as the Linux kernel. To achieve this goal, we implemented one PDR adaptation for software verification, and another approach that integrates a PDR-like invariant-generation module into a k-induction approach.

PDR Adaptation for Software Verification. PDR is a model-checking algorithm that tries to construct an inductive safety invariant by incrementally learning clauses that are inductive relative to previously learned clauses. The clause-learning strategy is guided by counterexamples to induction, i.e., each time a proof of inductiveness fails, the algorithm attempts to learn a new clause to avoid the same counterexample to induction in the future. Originally, this algorithm was designed as a SAT-based technique for Boolean finite-state systems. Every adaptation of PDR to software verification therefore needs to consider how to effectively and efficiently handle the infinite state space and how to transfer the algorithm from SAT to SMT. Furthermore, the adaptation to software has to deal with the program counter.

PDR-like Invariant Generation. Whenever an induction-proof attempt fails with a counterexample, the counterexample describes a state s that can transition into a bad state (that violates the safety property), which means that in order to make the proof succeed, s must be removed from consideration by an auxiliary invariant. From this bad-state predecessor s, the clause-learning strategy of PDR proceeds to generate such an auxiliary invariant by applying the following two steps: (1) s is first generalized to a set of states C that all transition into a bad state; (2) an invariant is constructed that is (a) inductive relative to previously found invariants[1] and (b) at least strong enough to eliminate all states in C. If it fails to construct such an invariant and prove its inductiveness, then the steps are recursively re-applied to the counterexample obtained from the failed induction attempt.

We experimentally investigate two implementations of adaptations of PDR to software verification (CPAchecker-CTIGAR and Vvt-CTIGAR), as well as several combinations that use the PDR-like invariant-generation module that we designed and implemented for this study.

Example. Figure 1 shows an example C program (`eq2.c`) that contains four unsigned integer variables `w`, `x`, `y`, and `z`. In line `10`, the variable `w` is initialized to an unknown value via the input function `__VERIFIER_nondet_uint()`; then, its value is copied to `x` in line `11`. In line `12`, variable `y` is initialized with the value of `w + 1`, and in line `13`, variable `z` is initialized with the value of `x + 1`, such

[1] An assertion F is said to be inductive relative to an invariant Inv if Inv can be used as an auxiliary invariant for the proof of inductiveness $\forall s_j, s_{j+1} : F(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow F(s_{j+1})$ by conjoining Inv to the induction hypothesis $F(s_j)$, such that the modified induction query $\forall s_j, s_{j+1} : F(s_j) \wedge \boldsymbol{Inv}(\mathbf{s_j}) \wedge T(s_j, s_{j+1}) \Rightarrow F(s_{j+1})$ allows a proof by induction to succeed. [12]

```
1  extern void __VERIFIER_error() __attribute__
↪  ((__noreturn__));
2  extern unsigned int __VERIFIER_nondet_uint(void);
3  void __VERIFIER_assert(int cond) {
4    if (!(cond)) {
5      ERROR: __VERIFIER_error();
6    }
7    return;
8  }
9  int main(void) {
10   unsigned int w = __VERIFIER_nondet_uint();
11   unsigned int x = w;
12   unsigned int y = w + 1;
13   unsigned int z = x + 1;
14   while (__VERIFIER_nondet_uint()) {
15     y++;
16     z++;
17   }
18   __VERIFIER_assert(y == z);
19   return 0;
20 }
```

Fig. 1: Example C program `eq2.c`

that at this point, `w` and `x` are equal to each other, and `y` and `z` are also equal to each other. Then, from line `14` to line `17`, a loop with a nondeterministic exit condition (and therefore an unknown number of iterations) increments in each iteration both variables `y` and `z`. Lastly, line `18` asserts that after the loop, `y` and `z` are (still) equal to each other. Since `y` and `z` are equal before the loop, and are always incremented together within the loop, the invariant $y = z$ is inductive. However, since there is no direct connection between `y` and `z` but only an indirect one via their shared dependency on `w`, naïve data-flow-based techniques may fail to find this invariant. In fact, we tried several configurations of the verification framework CPAchecker, and found that many of them fail to prove this program:

- Plain k-induction without auxiliary-invariant generation fails, because it never checks if $y = z$ is a loop invariant and instead only checks the reachability of the assertion failure (located after loop). The reachability of the assertion failure, in turn, depends on the nondeterministic loop-exit condition. Therefore we cannot conclude from "the assertion failure was not reached in k previous iterations" that "the assertion failure cannot be reached in the next iteration": In the absence of auxiliary invariants, a valid counterexample to this induction hypothesis would always be that in the previous iterations the assertion *condition* was in fact violated and an assertion *failure* was not reached only *because the loop was not exited*.
- A data-flow analysis based on the abstract domain of Boxes [21] fails, because it is not able to track variable equalities.
- A data-flow analysis based on a template Eq for tracking the equality of pairs of variables fails, because while it detects the invariant $w = x$, it is unable to

make the step to $y = z$ due to the inequalities between `w` and `y`, and `x` and `z`, respectively.
- For consistency with our evaluation, we also applied a data-flow analysis based on a template for tracking whether a variable is even or odd; obviously this is not useful for this program, and thus, this configuration also fails.
- Even combining the previous three techniques into a compound invariant generator that computes auxiliary invariants for k-induction does not yield a successful configuration for this verification task.
- The invariant generator KIPDR (the above-mentioned adaptation of PDR to k-induction, which we present in more detail in Sect. 3), however, detects the invariant $y = z$ and is therefore able to construct a proof by induction for this verification task.

We will now briefly sketch how KIPDR detects the invariant $y = z$ for the example verification task. At first, KIPDR attempts to prove by induction that when line `18` is reached, the assertion condition holds, which fails as discussed previously. However, this failed induction attempt yields a counterexample to induction where the values of `y` and `z` differ from each other, e.g., $y = 0 \wedge z = 1$, which is then generalized to $y \neq z$, i.e., a set of states that includes the concrete predecessor of a bad state from the counterexample, as well as many other states that would violate the assertion, if they were reachable themselves. Then, KIPDR attempts to find an inductive invariant that eliminates all of these states, and the attempt succeeds with the invariant $y = z$. Afterwards, KIPDR re-attempts its original induction proof to show that the assertion is never violated, which now succeeds due to the auxiliary invariant $y = z$.

Contributions. We present the following contributions:

- We implement one adaptation of PDR to software verification (based on [11, 20]) in the open-source verification framework CPAchecker, in order to establish a baseline for comparison with new ideas for improvement.
- We design and implement the algorithm KIPDR, as a new module for invariant generation that is based on ideas from PDR and use this module as an extension to a state-of-the-art approach to k-induction [5].
- We conduct a large experimental study to compare several tools and approaches to software verification using PDR as a component, to highlight strengths and weaknesses of PDR in the domain of software verification.
- We contribute a set of small examples that need invariants that are more difficult to obtain for standard data-flow-based approaches than the invariants necessary for programs in the large benchmark set.

Related Work. While PDR (also known as IC3 for its first implementation [12]) was introduced as a SAT-based algorithm for model checking finite-state Boolean transition systems [13], several approaches have since then been presented to extend it to SMT and to apply it to the verification of software models: PDR has been suggested as an interpolation engine for Impact, but experiments have shown that it is too expensive in the general case, and is most effective if only

applied as a fall-back engine for cases where a cheaper interpolation engine fails to produce useful interpolants [15]. It also has been proposed to improve this approach by tracking control-flow locations explicitly instead of symbolically [28], thereby avoiding the problem that many iterations of the algorithm are spent only to learn the control flow, and this idea has later been extended by several improvements to the generalization step of PDR [29]. Another approach is to model the program using a Boolean abstraction, which has the advantage that it requires only few changes to the original algorithm, but the disadvantage that a refinement procedure is necessary to handle the spurious paths introduced by the abstraction: One such approach uses infeasible error paths (i.e., counterexample-guided abstraction refinement (CEGAR) [17]) to refine the abstraction [16], while another (CTIGAR) uses counterexamples to induction [11]; both of these refinement techniques use interpolation to obtain abstraction predicates; the latter of the two techniques is used in two of the configurations we compare in our evaluation (CPAchecker-CTIGAR and Vvt-CTIGAR [20]). A different extension of PDR to verify infinite-state systems that does not require abstraction refinement is property-directed k-induction [25], which increases the power of the induction checks used in PDR by applying k-induction instead of 1-induction, and which uses model-based generalization in addition to interpolation to reason about potentially-infinite sets of states. Unfortunately, support for effective model-based generalization is rare in SMT solvers [2], making this approach impractical. In contrast, our KIPDR algorithm presented in Sect. 3 only requires support for interpolation, which is available in several SMT solvers.

Despite this multitude of adaptations of PDR to infinite-state systems, most implementations in practice require their input to be encoded as transition systems. The only available software verifiers applicable to actual C programs and implement PDR-based techniques are CPAchecker [7], SeaHorn [23], and Vvt [20].

2 Background

In this section, we briefly introduce the algorithms PDR and k-induction, which provide the core concepts on which we base our ideas. In the following description of PDR and k-induction, we use the following notation: given the state variables s and s' within a state-transition system T that represents the program, predicate $I(s)$ denotes that s is an initial state, $T(s, s')$ that a transition from s to s' exists, and $P(s)$ that the safety property P holds for state s.

2.1 PDR

PDR maintains a list of k frames, where a frame F_i is a predicate that represents an overapproximation of all states reachable within at most $0 \leq i \leq k$ steps, and a queue of proof obligations, which guide invariant discovery towards invariants

[2] The implementation of the approach of property-directed k-induction combines two SMT solvers, because neither of them supports all features required by the technique.

relevant to prove the correctness of a safety property P. For a given state s, the notation $F_i(s)$ means that the predicate F_i holds for state s. The index i of a frame F_i is called its *level*, and the frame F_k is called the *frontier*, because it represents the largest overapproximation of reachable states computed by the algorithm [12]. The algorithm maintains the following invariants:

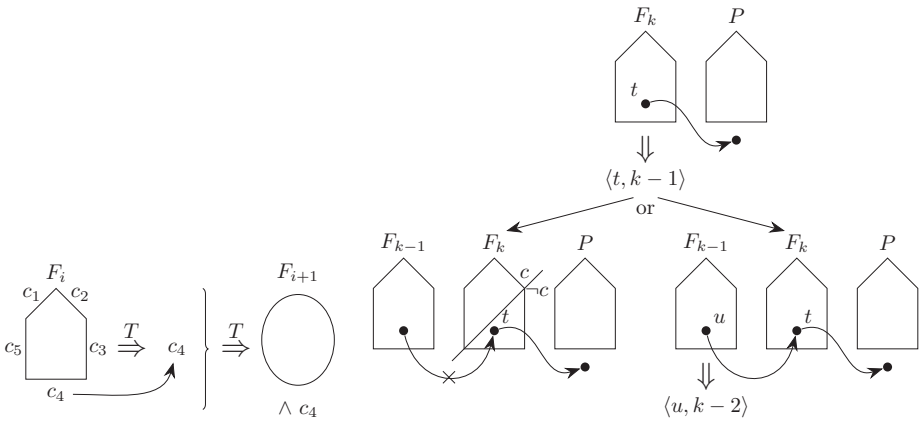
1. $F_0(s) = I(s)$, i.e., the first frame represents precisely the initial states.
2. $\forall i \in \{0, \ldots, k\} : F_i(s) \Rightarrow P(s)$, i.e., every frame contains only states that satisfy the safety property.
3. $\forall i \in \{0, \ldots, k-1\} : F_i(s) \Rightarrow F_{i+1}(s)$, i.e., a frame F_{i+1} represents in addition such states that are reachable with $i + 1$ steps.
4. $\forall i \in \{0, \ldots, k-1\} : F_i(s) \wedge T(s, s') \Rightarrow F_{i+1}(s')$, i.e., each frame is inductive relative to its predecessor.

Using these data structures and algorithm invariants, the algorithm attempts to find either a counterexample to P or a 1-inductive invariant F_i such that $F_i(s) \Leftrightarrow F_{i+1}(s)$ for some level $i \in \{0, \ldots, k-1\}$. Until either of these potential outcomes is reached, PDR shifts back and forth between the following two phases:

1. If the set of states represented by the frontier F_k does not contain any predecessor states of $\neg P$-states (i.e., $\forall s_j, s_{j+1} : F_k(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow P(s_{j+1})$, called frontier-incrementation check), a new frontier F_{k+1} is created and initialized to P. Subsequently, the algorithm attempts to push forward [3] each predicate c of each frame F_i with $0 \leq i \leq k$ for which the consecution check $F_i(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow c(s_{j+1})$ holds (see Fig. 2a). If, on the other hand, the frontier-incrementation check fails, PDR extracts a $\neg P$-predecessor t in F_k, which represents a counterexample to induction (CTI), from the failed query as proof obligation $\langle t, k-1 \rangle$ (see Fig. 2b, top).
2. While the queue of proof obligations is not empty, PDR processes the queue by trying to prove for each proof obligation $\langle t, i \rangle$ that the CTI-state t is itself not reachable from F_i and therefore does not need to be considered as a relevant $\neg P$-predecessor. For this proof, PDR chooses some predicate $c \Rightarrow \neg t$ with $\forall s : F_i(s) \Rightarrow c(s)$. PDR then checks if c is inductive relative to F_i by performing the consecution check $F_i(s_j) \wedge c(s_j) \wedge T(s_j, s_{j+1}) \Rightarrow c(s_{j+1})$. If the consecution check succeeds, the frames $F_1, \ldots, F_{i+1}$ can be strengthened by adding c, thus ruling out the CTI t in these frames for the future (see Fig. 2b, left). Also, unless $i = k$, we add a new proof obligation $\langle t, i+1 \rangle$ to the queue as an optimization to initiate forward propagation, because we expect that the CTI-state s would otherwise be rediscovered later at a higher level [11]. Otherwise, i.e., the consecution check does *not* succeed for clause c, the algorithm extracts a predecessor u of t from the failed consecution check, which is added as a new proof obligation $\langle u, i-1 \rangle$ if $i > 0$ and $t \wedge I$ is unsatisfiable (see Fig. 2b, right). Otherwise, u represents the initial state of a real counterexample to P.

An example of this algorithm is presented in a technical report [8, pp. 7–8]. A more detailed presentation of PDR can be found in the literature [12].

[3] By "push forward", we mean to add a predicate c from frame F_i to frame F_{i+1} [12].



(a) Consecution check makes sure to only conjoin to frame F_{i+1} such c_i from F_i that are inductive relative to F_i w.r.t. transition relation T

(b) If phase 1 results in a proof obligation $\langle t, k-1 \rangle$ (top), then phase 2 resolves either by strengthening F_k with c (left), or by creating a new (backwards) proof obligation $\langle u, k-2 \rangle$ (right); if the chain of proof obligations propagates back to the initial states, then a feasible error path is found

Fig. 2: Visualization of (a) the consecution check and (b) the handling of proof-obligations.

2.2 *k*-Induction

Like PDR, k-induction attempts to prove a safety property P by applying induction. However, while PDR strengthens its induction hypothesis by using clauses extracted from specific counterexamples to induction after failed induction attempts, k-induction strengthens its induction hypothesis by increasing the length of the unrolling of the transition relation.

Starting with an initial value for the bound k (usually 1), the k-induction algorithm increases the value of k iteratively after each unsuccessful attempt at finding a specification violation (base case), proving correctness via complete loop unrolling (forward condition), or inductively proving correctness of the program (inductive-step case).

Base Case. The base case of k-induction consists of running BMC with the current bound k.[4] This means that starting from all initial program states, all

[4] We define the loop bound as the number of visits of the loop head, that is, with loop bound $k = 1$, the loop head is visited once, but there was not yet any unwinding of the loop body. This nicely matches the intuition for k-induction: 1-inductiveness means that if the invariant holds for one state (without loop unrolling), then it holds again after one loop unrolling in the successor state; k-inductiveness means that if the invariant holds for k states ($k-1$ loop unrollings), then it holds again after one more loop unrolling in the successor state.

states of the program reachable within at most $k-1$ unwindings of the transition relation are explored. If a $\neg P$-state is found, the algorithm terminates.

Forward Condition. If no $\neg P$-state is found by the BMC in the base case, the algorithm continues by performing the forward-condition check, which attempts to prove that BMC fully explored the state space of the program by checking that no state with distance $k' > k-1$ to the initial state is reachable. If this check is successful, the algorithm terminates.

Inductive-Step Case. The forward-condition check, however, can only prove safety for programs with finite (and, in practice, short) loops. To prove safety beyond the bound k, the algorithm applies induction: The inductive-step case attempts to prove that after every sequence of k unrollings of the transition relation that did not reach a $\neg P$-state, there can also be no subsequent transition into a $\neg P$-state by unwinding the transition relation once more. In the realm of model checking of software, however, the safety property P is often not directly k-inductive for any value of k, thus causing the inductive-step-case check to fail. It is therefore state-of-the-art practice to add auxiliary invariants to this check to further strengthen the induction hypothesis and make it more likely to succeed. Thus, the inductive-step case proves a program safe if the following condition is unsatisfiable:

$$Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$$

where Inv is an auxiliary invariant, and $s_n, \ldots, s_{n+k}$ is any sequence of states. If this check fails, the induction attempt is inconclusive, and the program is neither proved safe nor unsafe yet with the current value of k and the given auxiliary invariant. In this case, the algorithm increases the value of k and starts over.

A detailed presentation of k-induction can be found in the literature [5, 6].

3 Combining k-Induction with PDR

Algorithm 1 shows an extension of k-induction with continuously-refined invariants [5] that applies PDR's aspect of learning from counterexamples to induction and that can be applied both as a main proof engine as well as an invariant generator. This allows us to apply this extension of k-induction as an invariant generator to a main k-induction procedure, similar to the KI⇄KI approach [5].

Inputs. The algorithm takes the following inputs: The value k_{init} is used to initialize the unrolling bound k, whereas the function inc is used to increase k in line 33 after each major iteration of the algorithm, up to an upper limit of k defined by the value k_{max} enforced in line 3. The set of initial program states is described by the predicate I, the possible state transitions are described

Algorithm 1 Iterative-Deepening k-Induction with Property Direction

Input: the initial value $k_{init} \geq 1$ for the bound k,
an upper limit k_{max} for the bound k,
a function $\mathsf{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \mathsf{inc}(n) > n$,
the initial states defined by the predicate I,
the transfer relation defined by the predicate T,
a safety property P,
a function get_currently_known_invariant to obtain auxiliary invariants,
a Boolean pd that enables or disables property direction,
a function $\mathsf{lift} : \mathbb{N} \times (S \to \mathbb{B}) \times (S \to \mathbb{B}) \times S \to (S \to \mathbb{B})$, and
a function $\mathsf{strengthen} : \mathbb{N} \times (S \to \mathbb{B}) \times (S \to \mathbb{B}) \to (S \to \mathbb{B})$,
where S is the set of program states.
Output: **true** if P holds, **false** otherwise
Variables: the current bound $k := k_{init}$,
the invariant $\mathit{InternalInv} := \mathit{true}$ computed by this algorithm internally, and
the set $O := \{\}$ of current proof obligations.

1: **while** $k \leq k_{max}$ **do**
2: $\quad O_{prev} := O$
3: $\quad O := \{\}$
4: $\quad \mathit{base\_case} := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left(\bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg P(s_n) \right)$
5: $\quad$ **if** $\mathsf{sat}(\mathit{base\_case})$ **then**
6: $\quad\quad$ **return false**
7: $\quad \mathit{forward\_condition} := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$
8: $\quad$ **if** $\neg\, \mathsf{sat}(\mathit{forward\_condition})$ **then**
9: $\quad\quad$ **return true**
10: $\quad$ **if** pd **then**
11: $\quad\quad$ **for each** $o \in O_{prev}$ **do**
12: $\quad\quad\quad \mathit{base\_case}_o := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left(\bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg o(s_n) \right)$
13: $\quad\quad\quad$ **if** $\mathsf{sat}(\mathit{base\_case}_o)$ **then**
14: $\quad\quad\quad\quad$ **return false**
15: $\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad \mathit{step\_case}_{o_n} := \bigwedge_{i=n}^{n+k-1} (o(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg o(s_{n+k})$
17: $\quad\quad\quad\quad \mathit{ExternalInv} := \mathsf{get\_currently\_known\_invariant}()$
18: $\quad\quad\quad\quad \mathit{Inv} := \mathit{InternalInv} \wedge \mathit{ExternalInv}$
19: $\quad\quad\quad\quad$ **if** $\mathsf{sat}(\mathit{Inv}(s_n) \wedge \mathit{step\_case}_{o_n})$ **then**
20: $\quad\quad\quad\quad\quad s_o :=$ satisfying predecessor state
21: $\quad\quad\quad\quad\quad O := O \cup \{\neg \mathsf{lift}(k, \mathit{Inv}, o, s_o)\}$
22: $\quad\quad\quad\quad$ **else**
23: $\quad\quad\quad\quad\quad \mathit{InternalInv} := \mathit{InternalInv} \wedge \mathsf{strengthen}(k, \mathit{Inv}, o)$
24: $\quad \mathit{step\_case}_n := \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$
25: $\quad \mathit{ExternalInv} := \mathsf{get\_currently\_known\_invariant}()$
26: $\quad \mathit{Inv} := \mathit{InternalInv} \wedge \mathit{ExternalInv}$
27: $\quad$ **if** $\mathsf{sat}(\mathit{Inv}(s_n) \wedge \mathit{step\_case}_n)$ **then**
28: $\quad\quad$ **if** pd **then**
29: $\quad\quad\quad s :=$ satisfying predecessor state
30: $\quad\quad\quad O := O \cup \{\neg \mathsf{lift}(k, \mathit{Inv}, P, s)\}$
31: $\quad$ **else**
32: $\quad\quad$ **return true**
33: $\quad k := \mathsf{inc}(k)$
34: **return unknown**

by the transition relation T, and the set of safe states is described by the safety property P. The accessor get_currently_known_invariant is used to obtain the strongest invariant currently available via a concurrently running (external) auxiliary-invariant generator. A Boolean flag *pd* (reminding of "property-directed") is used to control whether or not failed induction checks are used to guide the algorithm towards a sufficient strengthening of the safety property P to prove correctness; if *pd* is set to *false*, the algorithm behaves exactly like standard k-induction. Given a failed attempt to prove some candidate invariant Q [5] by induction, the function lift is used to obtain from a concrete counterexample-to-induction (CTI) state a set of CTI states described by a state predicate C. An implementation of the function lift needs to satisfy the condition that for a CTI $s \in S$ where S is the set of program states, $k \in \mathbb{N}$, $Inv \in (S \to \mathbb{B})$, $Q \in (S \to \mathbb{B})$, and $C = \mathsf{lift}(k, Inv, Q, s)$, the following holds: $C(s) \wedge \left(\forall s_n \in S : C(s_n) \;\Rightarrow\; Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (Q(s_i) \wedge T(s_i, s_{i+1})) \Rightarrow \neg Q(s_{n+k}) \right)$, which means that the CTI s must be an element of the set of states described by the resulting predicate C and that all states in this set must be CTIs, i.e., they need to be k-predecessors of $\neg Q$-states, or in other words, each state in the set of states described by the predicate C must reach some $\neg Q$-state via k unrollings of the transition relation T. We can implement lift using Craig interpolation [18, 30] between $A : s = s_n$ and $B : Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (Q(s_i) \wedge T(s_i, s_{i+1})) \Rightarrow \neg Q(s_{n+k})$, because s is a CTI, and therefore we know that $A \Rightarrow B$ holds. [6] Hence, the resulting interpolant satisfies the criteria for C to be a valid lifting of s according to the requirements towards the function lift as outlined above. The function strengthen is used to obtain for a k-inductive invariant a stronger k-inductive invariant, i.e., its result needs to imply the input invariant, and, just like the input invariant, it must not be violated within k loop iterations and must be k-inductive.

Algorithm. Lines 4 to 6 show the base-case check (BMC) and lines 7 to 9 show the forward-condition check, both as described in Sect. 2. If *pd* is set to *true*, lines 10 to 23 attempt to prove each proof obligation using k-induction: Lines 12 to 14 check the base case for a proof obligation o. If any violations of the proof obligation o are found, this means that a predecessor state of a $\neg P$-state, and thus, transitively, a $\neg P$-state, is reachable, so we return *false*. If, otherwise, no violation was found, lines 16 to 23 check the inductive-step case to prove o. [7] We strengthen the induction hypothesis of the step-case check by

[5] Depending on the step the algorithm is in, Q may be either the safety property P or a proof obligation o.

[6] The formula C is called Craig interpolant for two formulas A and B with $A \Rightarrow B$, if $A \Rightarrow C$, $C \Rightarrow B$, and all variables in C occur in both A and B.

[7] Note that we do not need to check the forward condition for proof obligations, because the forward condition is unrelated to the safety property and the proof obligations, and therefore only needs to be checked once in each major iteration (i.e., once after each increment of k).

conjoining auxiliary invariants from an external invariant generator (via a call to get_currently_known_invariant) and the auxiliary invariant computed internally from proof obligations that we successfully proved previously. If the step-case check for o is unsuccessful, we extract the resulting CTI state, lift it to a set of CTI states, and construct a new proof obligation so that we can later attempt to prove that these CTI states are unreachable. If, on the other hand, the step-case check for o is successful, we no longer track o in the set O of unproven proof obligations (this case corresponds to line 22). We could now directly use the proof obligation as an invariant, but instead, in line 23 we first try to strengthen it into a stronger invariant that removes even more unreachable states from future consideration before conjoining it to our internally computed auxiliary invariant. In our implementation, we implement strengthen by attempting to drop components from a (disjunctive) invariant and checking if the remaining clause is still inductive. In lines 24 to 32, we check the inductive-step case for the safety property P. This check is mostly analogous to the inductive-step case check for the proof obligations described above, except that if the check is successful, we immediately return *true*.

Note that Alg. 1 eagerly increases k, even if the set O of proof obligations is not empty. This heuristic prevents the PDR part from iterating through long chains of proof obligations, it rather delegates the unrolling to the k-induction part.

An in-depth discussion of a practical example of Alg. 1 is presented in a technical report [8, pp. 12–14].

4 Evaluation

In this section, we present an extensive experimental study on the effectiveness and efficiency of adaptations of PDR to software verification.

4.1 Compared Approaches

We use the following abbreviations to distinguish between the different techniques that we evaluated:

CTIGAR: CTIGAR [11] is an adaptation of PDR to software verification. Our evaluation compares two implementations of CTIGAR, namely Vvt-CTIGAR from the tool Vvt and our own implementation CPAchecker-CTIGAR. Vvt [20] also provides a configuration that runs a parallel portfolio combination of Vvt-CTIGAR and bounded model checking, which we call Vvt-Portfolio.

KI: KI [5] denotes the plain k-induction algorithm without property direction and without auxiliary invariants, i.e., we configure Alg. 1 such that $pd = \mathit{false}$ and get_currently_known_invariant() always returns *true*.

KIPDR: KIPDR denotes a configuration of Alg. 1 such that $pd = \mathit{true}$ and get_currently_known_invariant() always returns *true*, i.e., k-induction with property direction but without additional auxiliary-invariant generation. KIPDR is, like CTIGAR, an adaptation of PDR to software verification.

KI⇜DF: KI⇜DF [5] denotes a parallel combination of k-induction (without property direction) with a data-flow-based auxiliary-invariant generator that continuously supplies the k-induction procedure with invariants. Here, we configure Alg. 1 such that $pd = false$ and get_currently_known_invariant() always returns the most recent (strongest) invariant computed by the data-flow-based auxiliary-invariant generator.

KI⇜KIPDR: Similarly to KI⇜DF, KI⇜KIPDR denotes a parallel combination of k-induction with an auxiliary-invariant generator — in this case, KIPDR — that continuously supplies invariants to the k-induction procedure. Here, we configure one instance of Alg. 1 such that $pd = false$ and get_currently_known_invariant() always returns the most recent (strongest) invariant computed by KIPDR (a second instance of Alg. 1 that is configured such that $pd = true$ and get_currently_known_invariant() always returns *true*).

KI⇜DF;KIPDR KI⇜DF;KIPDR denotes a parallel combination of k-induction with an auxiliary-invariant generator that uses a sequential combination of a data-flow-based invariant generator and KIPDR to continuously supply k-induction with auxiliary invariants. We configure one instance of Alg. 1 such that $pd = false$ and get_currently_known_invariant() always returns the most recent (strongest) invariant computed by a sequential combination of the data-flow-based invariant generator and KIPDR (a second instance of Alg. 1 that runs after the invariant generator finishes and is configured such that $pd = true$ and get_currently_known_invariant() always returns *true*).

We do not evaluate the used invariant generators as standalone approaches, as they are designed specifically to be used as auxiliary components and do not perform well enough in isolation. For example, data-flow based invariant-generation approaches are often too imprecise to verify tasks, whereas more precise techniques like KIPDR might run into too many timeouts to be competitive. Instead, we use the framework of k-induction with continuously refined invariant generation, which has been shown to be able to combine quick and precise techniques [5].

4.2 Experimental Setup

Details about the experimental setup can be found in the technical report [8], which describes in Sect. 4.2 which tool versions and SMT theory we used, in Sect. 4.3 which benchmark sets we used and why, in Sect. 4.4 which existing verifiers we compared to and which versions we took, in Sect. 4.5 which computing resources and execution environment were used, in Sect. 4.6 the scoring schema, and in Sect. 4.12 which threats to the validity of the evaluation we identified and how we mitigated them.

4.3 Results

In the following, we pick a few highlights from the results of our experimental evaluation, in order to illustrate the potential of the approaches. A complete and more detailed report of the results is available in the extended version of this article [8].

Table 1: Results for all 5 591 verification tasks, 1 457 of which contain bugs, while the other 4 134 are considered to be safe, for the two CTIGAR implementations CPAchecker-CTIGAR and Vvt-CTIGAR, for a theoretical "virtual best" combination of both CTIGAR implementations where an oracle selects the best implementation for each task, for k-induction without auxiliary invariants (KI), and for the best configurations of each tool: CPAchecker's KI⟲DF;KIPDR, SeaHorn, and Vvt as a portfolio verifier.

Verifier	CTIGAR		KI	Best of each tool		
	CPAchecker	Vvt		KI⟲DF; KIPDR	SeaHorn	Vvt - Portfolio
Score	1 903	879	3 282	**5 398**	2 848	727
Correct results	1 087	739	2 075	3 095	3 468	839
Correct proofs	832	524	1 239	2 335	2 724	528
Correct alarms	255	215	836	760	744	311
Wrong proofs	**0**	5	**0**	**0**	46	9
Wrong alarms	1	14	2	2	117	22
Timeouts	3 982	110	2 764	2 006	1 476	524
Out of memory	23	28	315	243	231	22
Other inconclusive	498	4 695	435	245	253	4 175
Times for correct results						
Total CPU Time (h)	9.0	3.2	30	54	29	5.7
Mean CPU Time (s)	30	16	52	63	31	25
Median CPU Time (s)	4.9	0.24	9.8	10	0.89	0.45

Suitability of CPAchecker for PDR. The first set of experiments showed that our implementation is at least as good as (and even better than) the only available implementation of PDR for software model checking. Columns two and three of Table 1 compare the results obtained by running the two implementations of CTIGAR on the whole benchmark set, and the last column of the table shows the results achieved with the standard configuration of Vvt, which runs not only CTIGAR, but a portfolio analysis of CTIGAR and bounded model checking. The quantile plot in Fig. 3 shows the CPU times that the two tool configurations spent on their correct results.

KIPDR versus Data-Flow Techniques. Data-flow-based techniques are usually more efficient than KIPDR. The higher efficiency of data-flow-based techniques is most likely due to the simple form of the invariants needed to prove the programs correct. In order to experiment with progams that have some more interesting invariants, we created a few programs by hand and tried to verify those. Table 2 shows the results we obtained for these tasks. Our experiments support the hypothesis that KIPDR can be very strong and efficient on tasks that other approaches can not solve. It is important to note that this is an 'exists' statement and can not be generalized, as shown by the results that KIPDR is often outperformed by simpler, data-flow-based invariant-generation techniques.

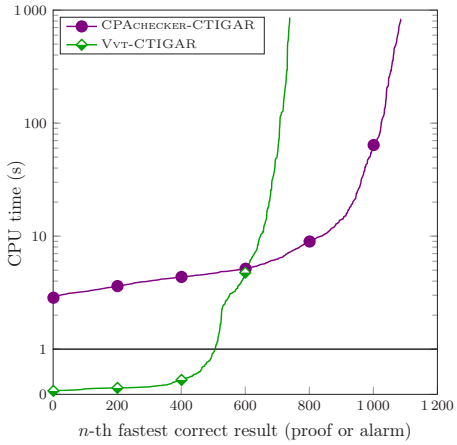


Fig. 3: Comparing two implementations of CTIGAR; quantile plot for accumulated number of solved tasks (proofs and alarms) showing the CPU time (linear scale below 1 s, logarithmic above) for the successful results of CPAchecker-CTIGAR and Vvt-CTIGAR

Table 2: Results of four k-induction-based configurations in CPAchecker with different approaches for generating auxiliary invariants for seven manually crafted verification tasks that do not contain bugs and are not solved by k-induction without auxiliary invariants; an entry "T" means that the CPU-time limit was exceeded, an entry "M" means that the memory limit was exceeded, and all other entries represent the CPU time a configuration spent to correctly solve the task

Task	KI←DF			KI←↺-KIPDR
	Boxes	Boxes, Eq	Boxes, Eq, Mod2	
const.c	3.3 s	3.3 s	**3.2 s**	3.8 s
eq1.c	T	**3.2 s**	3.3 s	4.9 s
eq2.c	M	M	M	**3.9 s**
even.c	T	T	**3.5 s**	3.9 s
odd.c	T	T	**3.4 s**	4.1 s
mod4.c	T	T	T	**3.6 s**
bin-suffix-5.c	M	M	M	**3.6 s**

Comparison with Non-PDR Approaches. The seven example programs [8] were added to the benchmark collection that was also used for SV-COMP 2019, and thus, results are available for all verifiers that participated in the competition [9]. Table 3 summarizes the results of the best six verifiers in comparison with the KI←↺-KIPDR approach that we created for the study in this paper. Those verifiers are, in alphabetical order, Skink, Ultimate Automizer, Ultimate Kojak,

[8] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp19/c/loop-invariants/

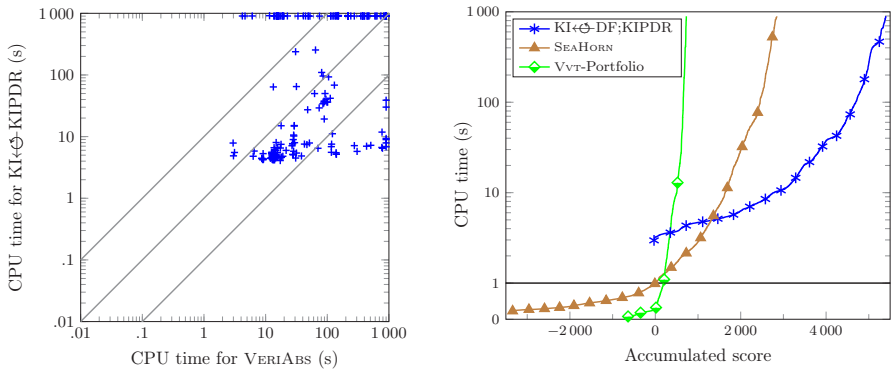
[9] See the last seven rows in this table: https://sv-comp.sosy-lab.org/2019/results/results-verified/ReachSafety-Loops.table.html

Table 3: Results of SV-COMP 2019 for the six verifiers that performed best on our seven manually crafted verification tasks, compared to the results of KI⇜⏲⇝KIPDR approach previously shown in Table 2; an entry "T" means that the CPU-time limit was exceeded, an entry "M" means that the memory limit was exceeded, an entry "O" means that the verifier gave up deliberately for other reasons, and all other entries represent the CPU time a verifier configuration spent to correctly solve the task; note that SV-COMP 2019 used Ubuntu 18.04 based on Linux 4.15, whereas our evaluation of KI⇜⏲⇝KIPDR used Ubuntu 16.04 based on Linux 4.4; otherwise, the evaluation environment was the same

Task	SV-COMP 2019						KI⇜⏲⇝KIPDR
	Skink	UAutomizer	UKojak	UTaipan	VeriAbs	VIAP	
`const.c`	4.2 s	8.7 s	9.1 s	8.2 s	13 s	110 s	**3.8 s**
`eq1.c`	290 s	7.8 s	7.6 s	8.3 s	14 s	57 s	**4.9 s**
`eq2.c`	4.1 s	8.1 s	8.6 s	7.6 s	14 s	4.7 s	**3.9 s**
`even.c`	**3.7 s**	7.4 s	8.2 s	8.6 s	140 s	4.5 s	3.9 s
`odd.c`	O	9.6 s	T	11 s	140 s	4.6 s	**4.1 s**
`mod4.c`	4.0 s	8.4 s	8.4 s	7.7 s	140 s	4.5 s	**3.6 s**
`bin-suffix-5.c`	O	14 s	T	13 s	13 s	4.7 s	**3.6 s**

Ultimate Taipan, VeriAbs, and VIAP. Fig. 4a directly compares the CPU times spent on tasks of in the subcategory *ReachSafety-Loops*, which is known to contain many tasks that require effort to be spent on generating loop invariants, by both VeriAbs, which was the best verifier in that subcategory, and KI⇜⏲⇝KIPDR. We observe that for the majority of tasks that were solved by both verifiers, KI⇜⏲⇝KIPDR is faster than VeriAbs, often by more than an order of magnitude. This shows that the invariant generator KIPDR can be significantly faster than other approaches, depending on the benchmark set. As before, a more in-depth discussion can be found in the technical report [8].

Comparison against PDR-Based Verification Tools. The last three columns of Table 1 give an overview over the best configurations of three software verifiers that use adaptations of PDR: For CPAchecker, we selected KI⇜⏲⇝DF;KIPDR. For SeaHorn, we used the same configuration as submitted by the developers to the 2016 Competition on Software Verification (SV-COMP 2016) [22]. For Vvt, we used the portfolio configuration. We observe that SeaHorn achieves the highest number of correct proofs, but also has a significant amount of incorrect proofs. CPAchecker is the slowest of the three tools and finds fewer proofs than SeaHorn, but CPAchecker has no wrong proofs, and also closely leads in the amount of found bugs. The score-based quantile plot of these results displayed in Fig. 4b visualizes the effects of incorrect results on the computed score. While the graph for SeaHorn is longer, i.e., shows that it solved the most tasks, it is offset to the left by a total penalty of −3 344 points, such that in the end, KI⇜⏲⇝DF;KIPDR accumulates the highest score because it has a smaller penalty of only −32 points.



(a) Scatter plot comparing the CPU times spent on tasks by VeriAbs and KI⟲-KIPDR

(b) Quantile plot for accumulated score of solved tasks (offset to the left by total penalty from wrong results) showing the CPU time (linear scale below 1 s, logarithmic above) for the successful results of KI⟲-DF;KIPDR, SeaHorn, and Vvt-Portfolio

Fig. 4: Plots that support the claim that the conclusions of the evaluation are relevant

These results confirm our hypothesis that our previous conclusions are relevant, because they are supported by an implementation that is competitive when compared to the best available PDR-based tool implementations.

5 Conclusion

Property-directed reachability (a.k.a. IC3) is a verification approach that is popular and successful in some fields of formal verification (e.g., hardware designs, Horn clauses). Unfortunately, there is a large gap between this success story and the applicability in practical software verification. We are closing this gap by (a) providing a well-engineered implementation of one published adaptation of PDR to software verification, (b) designing and implementing an invariant generator based on the ideas of PDR, and (c) providing an evaluation of all applicable tools and approaches on the largest available benchmark set of C verification tasks. This provides a good foundation as baseline for ongoing research in this area.

The results of our comparative evaluation extend the knowledge about PDR for software verification in the following ways: (1) Our implementation outperforms the existing implementation of PDR (Vvt) and is more precise than the other software verifier that uses PDR (SeaHorn). Thus, our implementation can serve as a reference implementation for further research on PDR for software verification. (2) On most of the programs in the widely used *sv-benchmarks* collection of verification tasks, other techniques are more effective (solve more problems) and more efficient (solve the problems faster). (3) PDR can be an effective and efficient technique for computing invariants that are difficult to obtain: there are programs for which our PDR-based approach is more efficient than the best invariant generator from SV-COMP in the subcategory *ReachSafety-Loops*.

5.1 Data Availability Statement

A replication package for this article including all evaluated implementations and BenchExec is available at Zenodo [9]. Current versions of CPAchecker are available at https://github.com/sosy-lab/cpachecker. The benchmark set of SV-COMP 2018 used in Sect. 4 is available online at https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp18 and the dataset from SV-COMP 2019 [4] that we analyzed is available at https://sv-comp.sosy-lab.org/2019/results/results-verified/All-Raw.zip.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Proc. TACAS. pp. 268–283. LNCS 2031, Springer (2001). https://doi.org/10.1007/3-540-45319-9_19
2. Ball, T., Rajamani, S.K.: The Slam project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). https://doi.org/10.1145/503272.503274
3. 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
4. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
5. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
6. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
7. 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
8. Beyer, D., Dangl, M.: Software verification with PDR: Implementation and empirical evaluation of the state of the art (August 2019), http://arxiv.org/abs/1908.06271
9. Beyer, D., Dangl, M.: Replication package for article 'Software verification with PDR: An implementation of the state of the art'. Zenodo (2020). https://doi.org/10.5281/zenodo.3678766
10. 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
11. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Proc. CAV. pp. 831–848. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_55
12. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI. pp. 70–87. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
13. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. **20**(4-5), 379–405 (2008). https://doi.org/10.1007/s00165-008-0080-9

14. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
15. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Proc. CAV. pp. 277–293. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_23
16. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. FMSD **49**(3), 190–218 (2016). https://doi.org/10.1007/s10703-016-0257-4
17. 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
18. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
19. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
20. Günther, H., Laarman, A., Weissenbacher, G.: Vienna Verification Tool: IC3 for parallel software (competition contribution). In: Proc. TACAS. pp. 954–957. LNCS 9636, Springer (2016)
21. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: Proc. SAS. pp. 287–303 (2010). https://doi.org/10.1007/978-3-642-15769-1_18
22. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: A framework for verifying C programs (competition contribution). In: Proc. TACAS. pp. 447–450. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_41
23. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
24. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
25. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Proc. FMCAD. pp. 85–92. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886665
26. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Proc. Int. Workshop on Parallel and Distributed Methods in Verification. pp. 55–62. EPTCS 72 (2011). https://doi.org/10.4204/EPTCS.72
27. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
28. Lange, T., Neuhäußer, M.R., Noll, T.: IC3 software model checking on control flow automata. In: Proc. FMCAD. pp. 97–104 (2015)
29. Lange, T., Prinz, F., Neuhäußer, M.R., Noll, T., Katoen, J.: Improving generalization in software IC3. In: Proc. SPIN'18. pp. 85–102. LNCS 10869, Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_5
30. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
31. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley Publishing, 3rd edn. (2011)
32. Wahl, T.: The k-induction principle (2013), available at `http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf`

Strategy Selection for Software Verification Based on Boolean Features
A Simple but Effective Approach

Dirk Beyer and Matthias Dangl

LMU Munich, Germany

Abstract. Software verification is the concept of determining, given an input program and a specification, whether the input program satisfies the specification or not. There are different strategies that can be used to approach the problem of software verification, but, according to comparative evaluations, none of the known strategies is superior over the others. Therefore, many tools for software verification leave the choice of which strategy to use up to the user, which is problematic because the user might not be an expert on strategy selection. In the past, several learning-based approaches were proposed in order to perform the strategy selection automatically. This automatic choice can be formalized by a strategy selector, which is a function that takes as input a model of the given program, and assigns a verification strategy. The goal of this paper is to identify a small set of program features that (1) can be statically determined for each input program in an efficient way and (2) sufficiently distinguishes the input programs such that a strategy selector for picking a particular verification strategy can be defined that outperforms every constant strategy selector. Our results can be used as a baseline for future comparisons, because our strategy selector is simple and easy to understand, while still powerful enough to outperform the individual strategies. We evaluate our feature set and strategy selector on a large set of 5 687 verification tasks and provide a replication package for comparative evaluation.

Keywords: Strategy selection · Software verification
Algorithm selection · Program analysis · Model checking

1 Introduction

The area of automatic software verification is a mature research area, with a large potential for adoption in industrial development practice. However, there are many usability issues that hinder the widespread use of the technology that is developed by researchers. One of the usability problems is that it is not explainable, for a given input program, which verification strategy to use. Different verification tools, algorithms, abstract domains, configurations, coexist with their different strengths in terms of approaching a verification problem.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11245, pp. 144–159, 2018.
https://doi.org/10.1007/978-3-030-03421-4_11

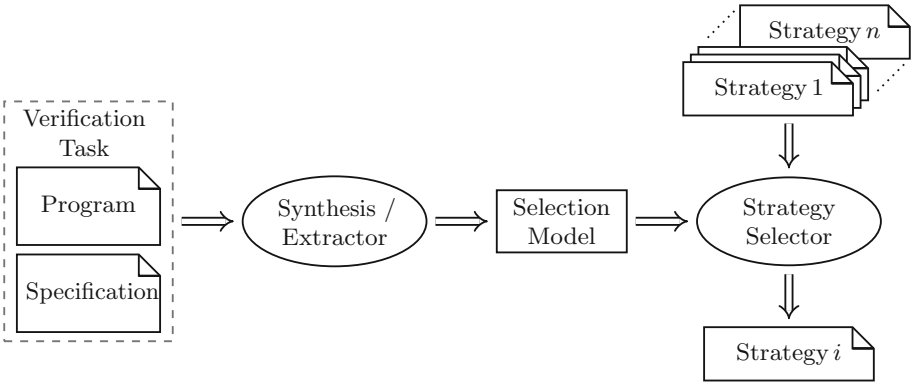


Fig. 1. Architecture of strategy selection (compare with Fig. 3 by Rice [28])

The insight that different verification techniques have different strengths was emphasized several times in the literature already. Most intensively, this can be derived from the results of the competition on software verification [4]. [1] A recent survey on SMT-based algorithms [6] (including bounded model checking, k-induction, predicate abstraction, and IMPACT) explains this insight concretely on specific example programs (from different categories of a well-known benchmark repository [2]): For each of the four considered algorithms, one example program is given that only this algorithm can efficiently verify and all other algorithms fail or timeout on this program. While there are powerful basic techniques, combinations or a selection are often a valuable strategy to further improve.

The problem has been understood for a long time in the research community, and there are several methods to approach the problem [24]. The standard techniques are sequential and parallel combinations [2, 7, 23, 26, 35]. These techniques are mostly based on statically assembling the combinations, and, by trying out one technique after the other (sequential) or by trying all at the same time (parallel), the problem is often solved by the approach that works best. However, there might be a considerable amount of resources wasted on unsuccessful computation work. For example, it might happen that one approach could solve the problem if all available resources were given to it, but since the resource is shared and assigned to several approaches, the overall verification does not succeed.

In order to solve this problem, a few techniques were proposed in the last few years that automatically select a potentially good verification strategy based on machine learning [18, 19, 20, 28, 32]. All those proposals share the common idea of strategy selection.

Strategy selection can be illustrated by the flow diagram in Fig. 1: The verification task (consisting of the source code of the input program and the

[1] https://sv-comp.sosy-lab.org/2018/results/results-verified/
[2] https://github.com/sosy-lab/sv-benchmarks

specification) is first analyzed and an abstract selection model is constructed (synthesized or extracted). The strategy selector predicts a strategy (from a given set of strategies) that should be used to solve the verification task, based on the information in the selection model.

The *selection model* can be either a vector of feature values, as defined by Rice's 'feature space' [28] (and implemented for software verification by, e.g., [19,20,32]), a graph representation of the program (e.g., [18]), or some other characteristics of the program and its specification. A selection model is *useful* if it contains sufficient information to distinguish verification tasks that need to be verified with different strategies. The model construction phase needs to extract the information from the source code of the input program and the specification. For example, the values of a feature vector might be extracted by static source-code measures.

The *set of strategies* (also called 'algorithm space' [28]) is either a set of algorithms, verification tools, different configurations of a configurable verification framework, or just a mere set of different parameter specifications for a single verifier.

The *strategy selector* is a function that takes as input a set of strategies and the selection model that represents some information about the program and its specification, and returns as output the strategy that is predicted to be useful to solve the verification task that is represented by the selection model.

Contributions. This paper makes the following contributions:

- We define a minimalist selection model, which (1) consists of an extremely small set of features that define the selection model and (2) a minimal range of values: all features are of type Boolean.
- We define an extremely simple strategy selector, which is based on insights from verification researchers.
- We implemented our feature measures and strategy selection in CPAchecker; the replication package contains all data for replicating the results.
- We perform a thorough experimental evaluation on a large benchmark set.

Related Work. We categorize the related work into the three areas of combinations, models, and machine learning.

Sequential and Parallel Combinations (Portfolios). While it seems obvious that combinations of strategies have a large potential, the topic was not yet systematically investigated in the area of software verification, while it has been used in other areas for many years [24,28]. One of the first ideas to combine different tools was for eliminating false alarms: after the core verifier has found an error path, this error path is not immediately reported to the user, but first converted into a program again which is then verified by an external verifier, and only if that external tool reports an error path as well, then the alarm is shown as a result to the user. [3]

[3] An early version of CPAchecker [9] had constructed a path program [8], dumped it to a file in C syntax, and then called Cbmc [17] as external verifier for validation. Meanwhile, such an error-path check is a standard component in many verifiers.

Other examples for sequential combinations are CPAchecker and SDV. CPAchecker [9] won the competition on software verification 2013 (SV-COMP'13, [3]) using a sequential combination [35] that started with explicit-state model checking for up to 100 s and then switched to a predicate analysis [10]. The static driver verification (SDV) [2] tool chain at Microsoft used a sequential combination (described in [32]) which first runs Corral [25] for up to 1 400 s and then Yogi [27].

Examples of parallel combinations are the verifiers Ufo [23] and PredatorHP [26], which start several different strategies simultaneously and take the result from the approach that terminates first.

Conditional model checking [7] is a technique to construct combinations with passing information from one verifier to the other. This technique can also be used to split programs into parts that can be independently verified [30].

Selection Models. A strategy selector needs a selection model of the program, in order to be able to classify the program and select a strategy. The classic way of abstracting is to define a set of features and the resulting vector of feature values is the selection model, which is in turn given to the strategy selector as input. There are various works on identifying features that are useful for classifying programs using its source code. Domain types [1] refine the integer types of C programs into more fine-grained integer types, in order to estimate what kind of abstract domain should be used to verify the program, for example, whether a BDD-based analysis or an SMT-based analysis is preferable. Variable roles [15,29,33,34] were used to analyze and understand programs, but also to classify program variables [22] according to how they are used in the program, i.e., what their role is. It has been shown that variable roles can help to determine which predicates should be used for predicate abstraction [21]. More sophisticated selection models can be used for machine-learning-based approaches. For example, one approach is based on graph representations of the program [18].

Machine-Learning-Based Approaches. The technique MUX [32] can be used to synthesize a strategy selector for a set of features of the input program and a given number of strategies. The strategies are verification tools in this case, and the feature values are statically extracted from the source code of the input program. Unfortunately, this technique is not reproducible, as reported by others [20]. Later, a technique that uses more sophisticated features was proposed [19,20]. While the above techniques use explicit features (defined by measures on the source code), a more recently developed technique [18] leaves it up to the machine learning to obtain insights from the input program. The advantage is that there is no need to define the features: the learner is given the control-flow graph, the data-dependency graph, and the abstract syntax tree, and automatically derives internally the characteristics that it needs. Also, the technique predicts a ranking, that is, the strategy selector is a function that maps verification tasks not to a single strategy, but to a sequence of strategies.

2 An Approach Based on Simple Boolean Features

Our goal is to define a strategy-selection approach that is simple and easy to understand but still effectively improves the overall performance.

2.1 Selection Model

We identify the following criteria from which we define our selection model:

- The model is based on features of the input program that are efficiently extractable from the program's source code using a simple static analysis.
- The model consists of a small set of features.
- The features have a small set of values.

Based on sets of program characteristics that were reported in the literature [1,22], we selected a few extremely coarse features. We will later evaluate whether our choice of features can instantiate a model that contains sufficient information to distinguish programs that should be verified by different strategies. Let $V = P \times S$ be the set of all verification tasks, each of which consists of a program from the set P and a specification from the set S, and let $\mathbb{B}$ be the set of Boolean values. We define the following four features for our selection model:

hasLoop : $V \to \mathbb{B}$ with
$\mathsf{hasLoop}((p, \cdot)) = \mathit{true}$ if program p has a loop, and *false* otherwise

hasFloat : $V \to \mathbb{B}$ with
$\mathsf{hasFloat}((p, \cdot)) = \mathit{true}$ if program p has a variable of a floating-point type (`float`, `double`, and `long double` in C), and *false* otherwise

hasArray : $V \to \mathbb{B}$ with
$\mathsf{hasArray}((p, \cdot)) = \mathit{true}$ if program p has a variable of an array type, and *false* otherwise

hasComposite : $V \to \mathbb{B}$ with
$\mathsf{hasComposite}((p, \cdot)) = \mathit{true}$ if program p has a variable of a composite type (`struct` and `union` in C), and *false* otherwise

For example, consider a program with a loop and only variables of integer type; the selection model would be the feature vector $(\mathit{true}, \mathit{false}, \mathit{false}, \mathit{false})$.

2.2 Strategies

For our example instantiation of a strategy-selection approach, we use different strategies from one verification framework. [4] We choose the software-verification framework CPAchecker as framework to configure our strategies, because it

[4] This has the advantage that the performance difference is not caused by the use of different programming languages, parser frontends, SMT solvers, libraries, but by the conceptual difference of the strategy (better internal validity). While it would be technically easy to extend the set of available strategies to other software verifiers, we already obtain promising results by just using different CPAchecker strategies.

consistently yielded good results in the competition on software verification (SV-COMP) [4], and we can actually also compare against CPA-Seq, the winning strategy that CPAchecker used in SV-COMP 2018.[5] Also, CPAchecker is highly configurable and provides a comprehensive set of algorithms and components to choose from (e.g., [6,11]) as well as a simple mechanism for sequential [35] and parallel composition [31]. The description of our three verification strategies will refer to the following components:[6]

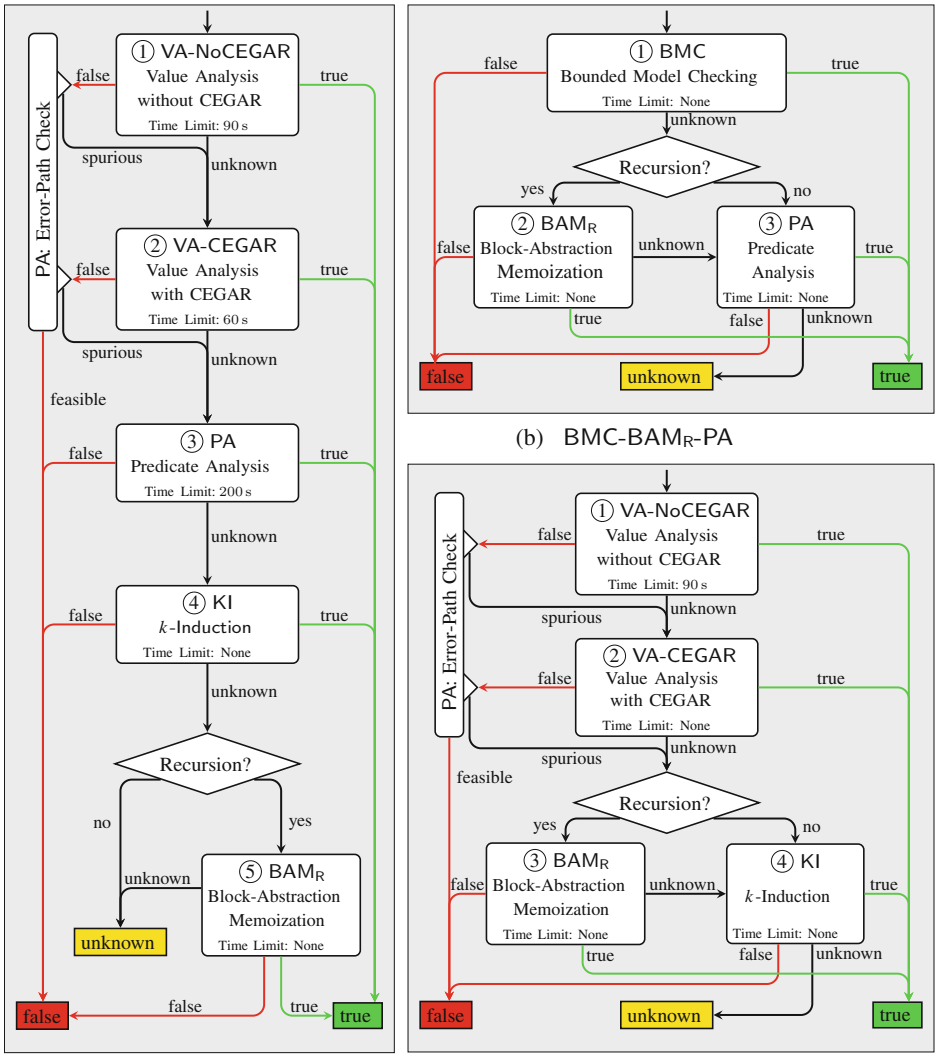
VA-NoCEGAR: value analysis without CEGAR[7] [11]
VA-CEGAR: value analysis with CEGAR [11]
PA: predicate analysis with CEGAR [10]
KI: k-induction with continuously refined invariant generation [5]
BAM$_\mathsf{R}$: block-abstraction memoization (BAM) [36] for a composite abstract domain of predicate analysis and value analysis
BMC: bounded model checking (BMC) [13]

The set of three verification strategies that we use in our strategy selector are the above mentioned strategy CPA-Seq that won the last competition and two more strategies that are based on components from the above list:

CPA-Seq is a sequential combination of VA-NoCEGAR, VA-CEGAR, PA, KI, and BAM$_\mathsf{R}$ as depicted in Fig. 2a: VA-NoCEGAR runs for up to 90 s, then VA-CEGAR runs for up to 60 s, then PA for up to 200 s, followed by KI for the remaining time. Any of the components may terminate early if it detects that it cannot handle the task. If none of the aforementioned components can handle the task and the last one (KI) fails because the task requires handling of recursion, the BAM$_\mathsf{R}$ component runs, which in our implementation is the only one that is able to handle recursion but lacks support for handling pointer aliasing and is therefore only desirable as a fallback for recursive tasks. If either VA-NoCEGAR or VA-CEGAR find a bug in the verification task, the error path is checked for feasibility with a PA-based error-path check; if the check passes, the bug is reported, otherwise, the component result is ignored and the next component runs.

BMC-BAM$_\mathsf{R}$-PA is a sequential combination of BMC, BAM$_\mathsf{R}$, and PA as depicted in Fig. 2b. As above, any of the components may terminate early if it detects that it cannot handle the task; otherwise there are no individual time limits for components in this strategy: As a result the first component of this strategy, BMC, runs until it solves the task or fails. If it fails because the task requires handling of recursion, the BAM$_\mathsf{R}$ component runs, with the same reasoning as for CPA-Seq; if the reason why bounded model checking failed was not recursion or if BAM$_\mathsf{R}$ also fails to solve the task, PA runs. This means that BAM$_\mathsf{R}$ and PA are only used as fallback components if the BMC component fails due to recursion or some other unsupported feature, whereas in all other cases, BMC would be the only component that runs.

[5] https://sv-comp.sosy-lab.org/2018/
[6] KI is, strictly speaking, already a composition, because it uses bounded model checking (BMC) [14] as a component.
[7] CEGAR is the abbreviation for counterexample-guided abstraction refinement [16].



(a) CPA-Seq

(b) BMC-BAM$_R$-PA

(c) VA-BAM$_R$-KI

Fig. 2. Sequential combinations of strategies

VA-BAM$_R$-KI is a sequential combination of VA-NoCEGAR, VA-CEGAR, BAM$_R$, and KI, as depicted in Fig. 2c. As above, any of the components may terminate early if it detects that it cannot handle the task; only the first component, VA-NoCEGAR, has an individual time limit and runs for up to 90 s. Afterwards, VA-CEGAR runs until it exceeds its time limit, fails, or solves the task. As in CPA-Seq, if either VA-NoCEGAR or VA-CEGAR find a bug in the verification task, the error path is checked for feasibility with a

PA-based error-path check; if the check passes, the bug is reported, otherwise, the component result is ignored and the next component runs. If VA-CEGAR fails because the task requires handling of recursion, the BAM_R component runs, with the same reasoning as for CPA-Seq; if the reason why VA-CEGAR failed was not recursion or if BAM_R also fails to solve the task, KI runs. This means that BAM_R and KI are only used as fallback components if VA-NoCEGAR and VA-CEGAR both fail due to recursion or some other unsupported feature, whereas in all other cases, either VA-NoCEGAR would solve the task within at most 90 s, or VA-CEGAR would attempt to solve it in the remaining time without switching to any further components.

2.3 Strategy Selector

Based on the three strategies and the selection model described above, we define our strategy selector Model-Based. Our strategy selector chooses the strategy based on the selection model as follows: It is defined to always choose the strategy $\mathsf{BMC\text{-}BAM}_\mathrm{R}\text{-}\mathsf{PA}$ if hasLoop is *false*, because if there is no loop, we do not need any potentially expensive invariant-generating algorithm. If hasLoop is *true*, and either of hasArray, hasFloat, or hasComposite is *true*, it chooses the strategy $\mathsf{VA\text{-}BAM}_\mathrm{R}\text{-}\mathsf{KI}$. If hasLoop is *true* and all of hasArray, hasFloat, and hasComposite are *false*, it chooses the strategy CPA-Seq:

$$\text{strategy} = \begin{cases} \mathsf{BMC\text{-}BAM}_\mathrm{R}\text{-}\mathsf{PA} & \text{if } \neg\mathsf{hasLoop} \\ \mathsf{VA\text{-}BAM}_\mathrm{R}\text{-}\mathsf{KI} & \text{if } \mathsf{hasLoop} \wedge (\mathsf{hasFloat} \vee \mathsf{hasArray} \vee \mathsf{hasComposite}) \\ \mathsf{CPA\text{-}Seq} & \text{otherwise} \end{cases}$$

While CPA-Seq consists of a wider variety of components that should in theory be more accurate for these complex features, $\mathsf{VA\text{-}BAM}_\mathrm{R}\text{-}\mathsf{KI}$, which consists mainly of value analysis, does not require expensive SMT solving and therefore often solves tasks where CPA-Seq exceeds the resource limitations.

3 Evaluation

In this section, we present an experimental study to compare the effectiveness of our approach to strategy selection to various fixed strategies (i.e., constant strategy selectors) and to serve as a baseline for future comparisons of potentially more elaborate approaches.

3.1 Evaluation Goals

The goal of our experimental evaluation is to confirm the following claims:

Claim 1: We claim that combining different strategies sequentially is more effective than each individual strategy by itself. To confirm this claim, we evaluate the composite strategy CPA-Seq as well as each of its individual components, and compare their results. For a successful confirmation, CPA-Seq must yield a higher score than each of its component strategies. If confirmed, this claim supports the insight that combinations should be used in practice.

Claim 2: We claim that by classifying a verification task using a small set of features and selecting a strategy to solve a task from a small set of verification strategies based on this classification, we can further improve effectiveness significantly. To confirm this claim, we evaluate three verification strategies individually, as well as two strategy selectors that can choose from the three sequential strategies: One of the strategy selectors will choose randomly, while the other one will base its choice on the selection model that we extracted from the task. To successfully show that strategy selection can improve effectiveness, the model-based strategy selector must yield a higher score than each of the individual strategies that it chooses from, and to show that the selection model is useful for the strategy selection, the model-based strategy selector must yield a higher score than the random strategy selector.

The random strategy selector Random that we need for Claim 2 chooses randomly with uniform distribution from the set of strategies, ignoring the selection model.

3.2 Benchmark Set

The set of verification tasks that we use in our experiments is taken from the benchmark collection that is also used in SV-COMP. In particular, we use all benchmark categories from SV-COMP 2018 [8] for which we have identified different strategies.

This means that we exclude the category *ConcurrencySafety* as well as the categories for verifying the properties for overflows, memory safety, and termination, for each of which there is only one known suitable strategy in CPAchecker. The remaining set of categories consists of 5 687 verification tasks from the subcategory *DeviceDriversLinux64_ReachSafety* of the category *SoftwareSystems* and from the following subcategories of the category *ReachSafety*: *Arrays*, *Bitvectors*, *ControlFlow*, *ECA*, *Floats*, *Heap*, *Loops*, *ProductLines*, *Recursive*, and *Sequentialized*. A total of 1 501 of these tasks are known to contain a specification violation, and we expect the other 4 186 to satisfy their specification.

3.3 Experimental Setup

For our experiments, we executed version `1.7.6-isola18` of CPAchecker on machines with one 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM each. The operating system was Ubuntu 16.04 (64 bit), using Linux 4.4 and OpenJDK 1.8. We limited each verification run to two CPU cores, a CPU run time of 15 min, and a memory usage of 15 GB. We used the benchmarking framework BenchExec [9] [12] to conduct our experiments, to ensure reliable and accurate measuremenfts.

[8] https://sv-comp.sosy-lab.org/2018/benchmarks.php
[9] https://github.com/sosy-lab/benchexec

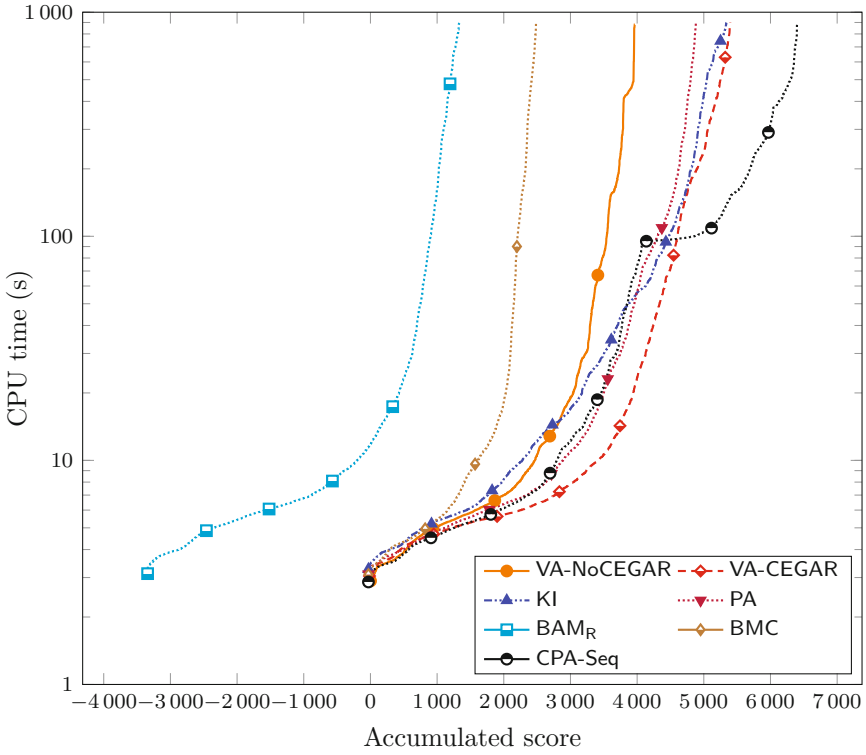


Fig. 3. Quantile functions of different individual strategies and one sequential combination of those strategies (CPA-Seq), as well as one further individual strategy (BMC), for their accumulated scores showing the CPU time for the successful results, offset to the left by the total penalty for incorrect results of each corresponding strategy

3.4 Presentation

The full results of our evaluation are available on a supplementary web page. [10] All reported times are rounded to two significant digits. To evaluate the choices of our strategy selector, we use the community-agreed scoring schema of SV-COMP, which assigns quality values to each verification result, i.e., we calculate a score that quantifies the quality of the results for a verification strategy. For every correct safety proof, 2 points are assigned and for every real bug found, 1 point is assigned. A score of 32 points is subtracted for every wrong proof of safety (false negative) and 16 points are subtracted for every wrong alarm (false positive) reported by the strategy, This scoring follows a community consensus [4] on the difficulty of verification versus falsification and the importance of correct results, and is designed to value safety higher than finding bugs, and to punish wrong answers severely.

[10] https://www.sosy-lab.org/research/strategy-selection/

Table 1. Results for all 5 687 verification tasks (1 501 contain a bug, 4 186 are correct), for all basic strategies

Approach	VA-NoCEGAR	VA-CEGAR	PA	KI	BAM_R	BMC
Score	3 966	5 397	4 881	5 340	1 335	2 484
Correct results	2 365	3 046	2 840	3 053	2 575	1 757
Correct proofs	1 601	2 367	2 073	2 319	2 104	759
Correct alarms	764	679	767	734	471	998
Wrong proofs	0	0	0	0	10	0
Wrong alarms	0	1	2	2	189	2
Timeouts	2 376	1 554	2 497	2 236	2 167	3 379
Out of memory	1	1	14	243	128	381
Other inconclusive	945	1 085	334	153	618	168
Times for correct results						
Total CPU Time (h)	30	54	39	68	33	28
Avg. CPU Time (s)	45	64	49	80	46	57
Total Wall Time (h)	24	44	33	43	29	25
Avg. Wall Time (s)	36	52	42	51	40	51

Table 2. Results for all 5 687 verification tasks (1 501 contain a bug, 4 186 are correct), for all combinations of basic strategies: simple sequential combinations, random choice between the sequential combinations, model-based strategy selection, and an imaginary oracle that always selects the best of the three strategies for any given task.

Approach	Sequential Combinations			Random	Model-Based	Oracle
	CPA-Seq	BMC-BAM_R-PA	VA-BAM_R-KI			
Score	6 399	2 612	6 442	5 174	6 790	7 036
% of Oracle Score	91	37	92	74	97	100
Correct results	3 740	1 840	3 740	3 122	3 932	4 111
Correct proofs	2 691	804	2 734	2 084	2 922	2 957
Correct alarms	1 049	1 036	1 006	1 038	1 010	1 154
Wrong proofs	0	0	0	0	0	0
Wrong alarms	2	2	2	2	4	2
Timeouts	1 715	3 385	1 879	2 317	1 486	1 347
Out of memory	194	406	26	202	224	185
Other inconclusive	36	54	40	44	41	42
Times for correct results						
Total CPU Time (h)	79	28	87	66	99	96
Avg. CPU Time (s)	76	54	83	76	90	84
Total Wall Time (h)	65	25	70	55	80	79
Avg. Wall Time (s)	63	48	67	63	73	69

3.5 Claim 1: Combining Strategies is Effective

In our first experiment we confirm the common knowledge that a sequential combination of several basic strategies can be more effective than either of its components. For this experiment, we compare the verification results of the winning strategy of the 7th Intl. Competition on Software Verification "CPA-Seq", to the results obtained by the basic strategies that it is composed of. Figure 3 shows the quantile functions for these strategies and Table 1 displays the detailed verification results and times for all basic strategies, whereas Table 2 contains the corresponding data for CPA-Seq and other combinations of strategies. We observe that CPA-Seq clearly outperforms the other strategies used in this experiment, even though it is only a sequential combination of the other strategies and contains no added features. We make the same observation for VA-BAM_R-KI, which is better than each of VA-NoCEGAR, VA-CEGAR, BAM_R, and KI. While BMC-BAM_R-PA is better than its main component BMC and its fallback component for recursion, BAM_R, it has a lower score than its other fallback component PA. The large amount of incorrect results produced by BAM_R and the resulting low score is caused by the lack of support for pointer-alias handling of this component mentioned in the description of strategies in Sect. 2.2, but while it is obviously unsuitable as a standalone strategy, it does add value as a fallback solution for CPA-Seq.

3.6 Claim 2: Strategy Selection is Effective

In our second experiment, we show that (1) using a strategy selector can be more effective than always choosing the same strategy. This is shown by the model-based strategy selector Model-Based, which achieves a higher score than each of the three strategies that it chooses from (compare column Model-Based with the columns CPA-Seq, BMC-BAM_R-PA, and VA-BAM_R-KI). Even the strategy selector Random performs better than one of the strategies that it chooses from (compare column BMC-BAM_R-PA with column Random).

We also show that (2) using our proposed selection model (consisting of a few simple Boolean features) is effective, because the strategy selector based on that model is more effective than a random choice between the three strategies, and also, for all three available choices, more effective than any constant strategy selector (always choosing the same strategy).

As we can see in Fig. 4, this model-based strategy selection pays off and yields a significantly higher score than each of its competitors. Table 2 shows that while this model-based strategy selection still offers room for improvement because it causes two more wrong alarms than the next-best strategy, this drawback is outweighed by the large amount of correct proofs it produces. This shows that even with a very simple set of Boolean features and a very small set of choices, we can already obtain very promising results. Due to the nature of this approach, adding more features to improve the granularity of the classification and adding more strategy choices to take advantage of the ability to complement this fine-grained classification with a better strategy for each class of tasks, can further improve upon our results. Table 2 also contains the column Oracle that

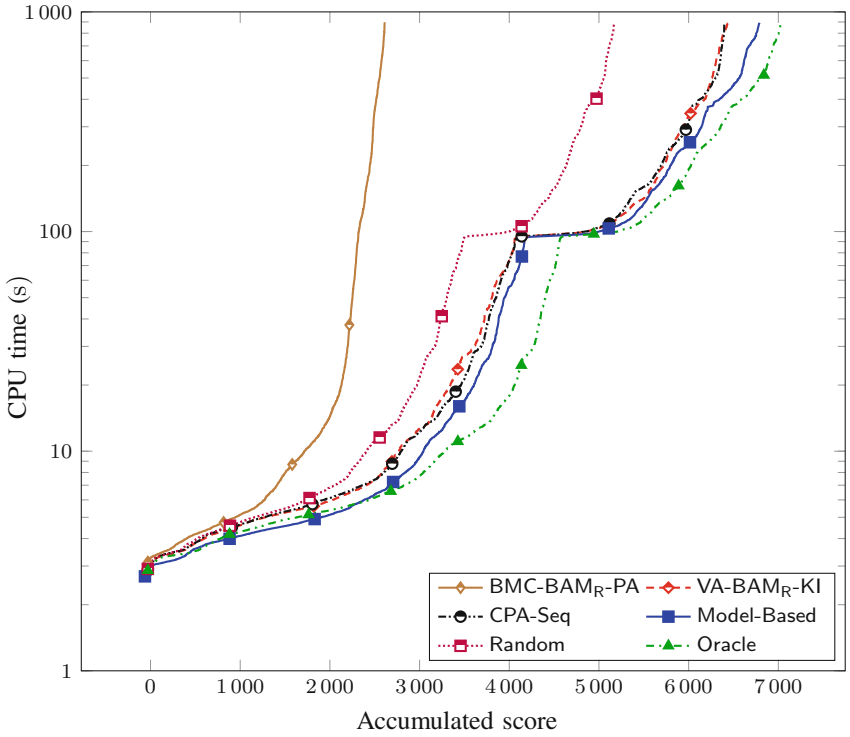


Fig. 4. Quantile functions of three different constant strategy selectors, one model-based strategy selector, one random strategy selector, and one selector based on a hypothetical all-knowing oracle, for their accumulated scores showing the CPU time for the successful results, offset to the left by the total penalty for incorrect results of each corresponding strategy

shows the best results obtainable by an (imaginary) ideal strategy selector based on an oracle that is able to determine the best of the three strategies CPA-Seq, BMC-BAM$_R$-PA, and VA-BAM$_R$-KI for each task, which achieves only 246 more points than our model-based selector. This means that our model-based selector reaches 97 % of the maximum score achievable by selecting between CPA-Seq, BMC-BAM$_R$-PA, and VA-BAM$_R$-KI on tasks of our benchmark set.

3.7 Threats to Validity

External Validity. Approaches for strategy selection that are not based on unsupervised learning are dependent on the strategies in the image range that the selector maps to. Therefore, our concrete instantiation of the selector is limited to the chosen strategies and does not consider other strategies of CPAchecker or other software verifiers.

We only showed that our selection model is useful for the given benchmark set. The benchmark set is taken from the largest and most diverse set of verification tasks that is publicly available, but the selection model might not sufficiently well distinguish verification tasks that are different from those in the benchmark set.

Note also that we considered only one verification property in the selection of the benchmark set and in the strategy selector. For benchmark sets with more than one verification property, it may be beneficial to define a strategy selector that considers the verification property as an additional feature to distinguish between tasks.

While the scoring schema from SV-COMP, which we used to model quality, is community agreed and quite stable in its design, a different scoring schema might favor a different strategy-selection function.

Internal Validity. While we used one of the best available benchmarking frameworks, namely BenchExec [11] [12], which is used by several international competitions, to conduct our experiments and ensure reliable and accurate measurements, there still might be measurement errors.

4 Conclusion

This paper explains an approach for strategy selection that is based on a simple selection model —a small set of Boolean features— that is easy to extract statically from the program source code. As strategies to choose from we use the winner of the last competition on software verification (SV-COMP'18) and two more strategies that we constructed from the same verification framework. We evaluated our approach to strategy selection on a benchmark set consisting of 5 687 verification tasks and show that our strategy selector outperforms the winner of the last competition. We hope that this result can be taken as a baseline for comparison of more sophisticated approaches to strategy selection.

References

1. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., Rhein, A.v.: Domain types: Abstract-domain selection based on variable usage. In: Proc. HVC. LNCS, vol. 8244, pp. 262–278. Springer (2013)
2. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Proc. FMCAD, pp. 35–42. IEEE (2010)
3. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. LNCS, vol. 7795, pp. 594–609. Springer (2013)
4. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS. LNCS, vol. 10206, pp. 331–349. Springer (2017)
5. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. LNCS, vol. 9206, pp. 622–640. Springer (2015)
6. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018)

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

7. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE, pp. 57:1–57:11. ACM (2012)
8. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Proc. PLDI, pp. 300–309. ACM (2007)
9. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. LNCS, vol. 6806, pp. 184–190. Springer (2011)
10. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
11. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. LNCS, vol. 7793, pp. 146–162. Springer (2013)
12. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer (2017)
13. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
14. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
15. Bishop, C., Johnson, C.G.: Assessing roles of variables by program analysis. In: Proc. CSEIT, pp. 131–136. TUCS (2005)
16. 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)
17. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
18. Czech, M., Hüllermeier, E., Jakobs, M., Wehrheim, H.: Predicting rankings of software verification tools. In: Proc. SWAN, pp. 23–26. ACM (2017)
19. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. In: Proc. CAV. LNCS, vol. 9206, pp. 561–579. Springer (2015)
20. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. Form. Methods Syst. Des. **50**(2-3), 289–316 (2017)
21. Demyanova, Y., Rümmer, P., Zuleger, F.: Systematic predicate abstraction using variable roles. In: Proc. NFM. LNCS, vol. 10227, pp. 265–281 (2017)
22. Demyanova, Y., Veith, H., Zuleger, F.: On the concept of variable roles and its use in software analysis. In: Proc. FMCAD, pp. 226–230. IEEE (2013)
23. Gurfinkel, A., Albarghouthi, A., Chaki, S., Li, Y., Chechik, M.: Ufo: Verification with interpolants and abstract interpretation (competition contribution). In: Proc. TACAS. LNCS, vol. 7795, pp. 637–640. Springer (2013)
24. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(7), 51–54 (1997)
25. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Proc. CAV. LNCS, vol. 7358, pp. 427–443. Springer (2012)
26. Müller, P., Peringer, P., Vojnar, T.: Predator hunting party (competition contribution). In: Proc. TACAS. LNCS, vol. 9035, pp. 443–446. Springer (2015)
27. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The yogiproject: Software property checking via static analysis and testing. In: Proc. TACAS. LNCS, vol. 5505, pp. 178–181. Springer (2009)

28. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976)
29. Sajaniemi, J.: An empirical analysis of roles of variables in novice-level procedural programs. In: Proc. HCC, pp. 37–39. IEEE (2002)
30. Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Beyer, D., Huisman, M. (eds.) Proc. TACAS, Part II. LNCS, vol. 10806, pp. 249–265. Springer (2018)
31. Stieglmaier, T.: Augmenting predicate analysis with auxiliary invariants. Master's Thesis, University of Passau, Software Systems Lab (2016)
32. Tulsian, V., Kanade, A., Kumar, R., Lal, A., Nori, A.V.: MUX: Algorithm selection for software model checkers. In: Proc. MSR. ACM (2014)
33. van Deursen, A., Moonen, L.: Type inference for COBOL systems. In: Proc. WCRE, pp. 220–230. IEEE (1998)
34. van Deursen, A., Moonen, L.: Understanding COBOL systems using inferred types. In: Proc. IWPC, pp. 74–81. IEEE (1999)
35. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS. LNCS, vol. 7795, pp. 613–615. Springer (2013)
36. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Proc. ICFEM. LNCS, vol. 7635, pp. 332–347. Springer (2012)

Verification Witnesses

DIRK BEYER and MATTHIAS DANGL, LMU Munich
DANIEL DIETSCH and MATTHIAS HEIZMANN, University of Freiburg
THOMAS LEMBERGER, LMU Munich
MICHAEL TAUTSCHNIG, Queen Mary University of London

Over the last years, witness-based validation of verification results has become an established practice in software verification: An independent validator re-establishes verification results of a software verifier using verification witnesses, which are stored in a standardized exchange format. In addition to validation, such exchangable information about proofs and alarms found by a verifier can be shared across verification tools, and users can apply independent third-party tools to visualize and explore witnesses to help them comprehend the causes of bugs or the reasons why a given program is correct. To achieve the goal of making verification results more accessible to engineers, it is necessary to consider witnesses as first-class exchangeable objects, stored independently from the source code and checked independently from the verifier that produced them, respecting the important principle of separation of concerns. We present the conceptual principles of verification witnesses, give a description of how to use them, provide a technical specification of the exchange format for witnesses, and perform an extensive experimental study on the application of witness-based result validation, using the validators CPAchecker, UAutomizer, CPA-witness2test, and FShell-witness2test.

CCS Concepts: • **Software and its engineering → Formal language definitions**; **Formal methods**; **Formal software verification**; • **Theory of computation → Automated reasoning**; **Program reasoning;**

Additional Key Words and Phrases: Violation witness, correctness witness, witness validation, software verification, program analysis, model checking, data-flow analysis, formal methods, certifying algorithm

ACM Reference format:
Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. 2022. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 57 (September 2022), 69 pages.
https://doi.org/10.1145/3477579

1 INTRODUCTION

The omnipresent dependency on software in society and industry makes it necessary to ensure a reliable and correct functioning of the software. This trend will continue and become even more important in the future. During the last decade, various conceptual breakthroughs in verification research were achieved, and, as showcased by the annual TACAS International Competition

This article builds on concepts and techniques that were introduced in previous articles: Proc. FSE 2015 [25], Proc. FSE 2016 [23], and Proc. TAP 2018 [26].
This work was funded in part by the **Deutsche Forschungsgesellschaft** (**DFG**) — 418257054 (Coop).
Authors' addresses: D. Beyer, M. Dangl, and T. Lemberger, LMU Munich, Oettingenstraße 67, Munich, 80538, Bayern, Germany; emails: beyer@sosy.ifi.lmu.de, dangl@sosy.ifi.lmu.de, lemberger@sosy.ifi.lmu.de; D. Dietsch and M. Heizmann, University of Freiburg, Georges-Köhler-Allee 52, Freiburg, 79110, Baden-Württemberg, Germany; emails: dietsch@informatik.uni-freiburg.de, heizmann@informatik.uni-freiburg.de; M. Tautschnig, Queen Mary University of London, Mile End Road, London, E1 4NS, United Kingdom; email: michael.tautschnig@qmul.ac.uk.

1049-331X/2022/09-ART57
https://doi.org/10.1145/3477579

on Software Verification (SV-COMP 2012–2020)[1] [9–13], many successful software verifiers are already available.

Despite the success stories of software verification in academia and industry [5, 55, 63, 100], the wide adoption of verification technology in software-development industry is still slow. There are several frequently mentioned reasons for this disconnect between engineering theory and practice [2]. One concern is connected with wrong results: Sometimes the verification result contradicts the expectation of the developer, either because a bug in the verification tool causes a genuinely incorrect result, or because of difficult-to-understand technicalities (e.g., caused by an inaccurate or vague specification) dismissed as unrealistic by the developer. In both cases, the developers lose trust in the verification results and consider the effort spent on investigating these results too expensive. Another concern is that even if the verification result matches the expectation, no value is added to the software-engineering process if the developer does not understand the verification result, or the result is not helpful for improving the software system.

Moreover, it is unlikely that future software verifiers are going to be more trustworthy, thereby resolving the issue of lack of trust in verification results. Software-verification systems constitute complex software, often with many known flaws and presumably many more unknown flaws. Future, more efficient and effective verification systems with more complex features can be expected to exacerbate this problem. Consider, for example, that scientists are exploring the application of machine learning to formal verification, with the goal to develop new verification tools by automatically learning rules for formally analyzing systems from large datasets [48]. While experiences in other fields, such as image recognition, suggest that machine learning may be a promising solution for tasks that were previously considered to be too complex for machines to solve, they also reveal weaknesses: The decision process of deep neural networks is often incomprehensible, and experiments have shown that they sometimes exhibit a significant lack of robustness [93]. If applied to formal verification, these techniques may therefore even amplify the previously outlined difficulties of understanding and trusting verification results.

Thus, it is imperative to require verification results to conform to an established, machine-readable, and exchangeable standard format that can be used to store, compare, explain, visualize, and validate verification results. For this purpose, we present the community-agreed standard exchange format for verification witnesses that was designed with these goals in mind, has been implemented in over 30 different software verifiers [11, 12], and has been used for visualization [22, 114] and validation [23, 25] of verification results. Using a validation step after verification enables a whole new area of verification research, enabling approximate verification algorithms (such as based on neural networks, or run on approximate hardware) because the imprecision is not problematic if all verification results are *validated* in a second phase of the verification work flow.

Violation witnesses [25] are verification witnesses that document bugs detected by a verifier and address the problem of false alarms that imprecise verification tools sometimes produce: Formerly, a verification tool reported found bugs as error traces in a tool-specific manner; those bug reports were often difficult to read and understand, and therefore hardly usable. As a consequence, determining whether the reported bug was a false alarm that could be ignored by the developer or described as an actual programming error that needed to be fixed was a tedious manual process. Exchangeable violation witnesses resolve this issue, because several validators have been developed to use these witnesses to validate verification results [25, 26, 44, 122] and the general syntax of the exchange format allows new tools for presentation to be developed and used [22, 114].

Correctness witnesses [23] are verification witnesses that describe proofs found by a verifier and address the problem that proofs are sometimes incomprehensible or, for unsound verification tools,

[1] http://sv-comp.sosy-lab.org/.

even wrong. Formerly, many verification tools did not report any auxiliary information about their proofs, while others output only algorithm- and implementation-specific proof data, such as SMT queries, that are a tool-specific aid for checking the consistency of some of the intermediate steps that lead to the reported result. But because the verifiers usually provided no means of validating the translation of the original verification task into the tool-specific model, the reports could not serve as a certificate for the correctness of the verification result. Exchangeable proof witnesses resolve this issue, because several validators have been developed to use these witnesses to re-establish the proof of correctness [23, 44].

Verification witnesses should be considered as first-class verification objects that have much more value than the plain verification result TRUE or FALSE. A verification result should only be trusted if a *reason* for the result is provided and the result can be re-established with the additional information. Therefore, we require a verifier to augment a verification result with an exchangeable and machine-readable verification witness, such that both claims of correctness and bug alarms may be validated. With this technique, a trusted validator can establish trust in the verification results produced by an untrusted verifier, and even in the absence of a trusted validator a user's confidence in a verification result can be increased by applying different, independent validators to a verification witness. The process of witness validation is fully automatic. Witnesses can also be read by humans (perhaps using an inspection or visualization tool [22, 114]).

One example of the practical application of witness validation is the annual TACAS International Competition on Software Verification (SV-COMP), which for the last few years (since 2015 [10]) has used witness-based result validation as an integral component of the scoring process: Full points are only awarded for a verification result that is accompanied by a verification witness that helped an independent validator to confirm (re-establish) the verification result. This rule may be one of the incentives that caused tool developers to improve the precision and soundness of their competition submissions over the last years, even though the direct score penalties for incorrect results have not been increased: In 2016, ten out of 13 participating verifiers in the category "Overall" reported false alarms for more than ten tasks that were known to be safe, one submission even claimed safety for 962 out of 2,348 verification tasks that were known to contain a bug, another submission claimed safety for 872 tasks known to contain a bug, and a third submission claimed safety for 336 tasks known to contain a bug [11]. In 2018, the second year after the introduction of correctness witnesses, on the other hand, only four out of 14 participating verifiers in the category "Overall" reported bugs for more than ten tasks known to be safe, and the highest amount of incorrect claims of safety reported by any of these submissions was 21.

This article discusses the conceptual principles of verification witnesses, presents four different violation-witness-based result validators and two correctness-witness-based result validators, and provides a technical specification of the common format for exchangeable witnesses. On the syntactic level, we use XML, more specifically GraphML [51], as a language to represent verification witnesses. On the semantic level, we use the standard concept of (non-deterministic) finite automata to represent verification witnesses. To demonstrate the practical applicability of verification witnesses for witness-based result validation, we perform an extensive experimental study using the validators CPAchecker, UAutomizer, CPA-witness2test, and FShell-witness2test. As such, this article expands on the authors' work on verification witnesses previously published in three conference articles [23, 25, 26] by (1) unifying the different aspects of verification witnesses that were presented in isolation in the conference articles, (2) adding an in-depth discussion of the conceptual background, (3) formally defining the involved concepts in more detail, (4) illustrating all presented approaches using a common running example, and (5) providing a significantly extended thorough experimental evaluation using updated implementations and benchmarks.

2 BACKGROUND

In this section, we introduce the basic concepts on which verification witnesses are based.

2.1 Program Representation Using Control-flow Automata

We restrict our presentation to a simple imperative programming language that contains only assignment, assumption declaration, function-call, and function-return operations, and where all program variables range over integers ($\mathbb{Z}$). We implemented our concepts in the tools CPAchecker [38], CPA-witness2test [26], FShell-witness2test [26], and UAutomizer [83, 84], all of which support C programs. We use **control-flow automata** (**CFA**) to represent programs [30]. A *control-flow automaton* (L, l_{init}, G) consists of a finite set L of program locations that model the program counter (to relate the CFA to the source code, we denote the program location before the operation on line i as l_i), the initial program location l_{init} (program entry), and a set $G \subseteq L \times Ops \times L$ of control-flow edges, each of which models an operation op of the set Ops of program operations that is executed during the flow of control from one program location to another. All variables that occur in an operation $op \in Ops$ are contained in the set X of program variables. A *variable assignment* $v : X \to \mathbb{Z}$ is a mapping that assigns to each variable from X a value from $\mathbb{Z}$.

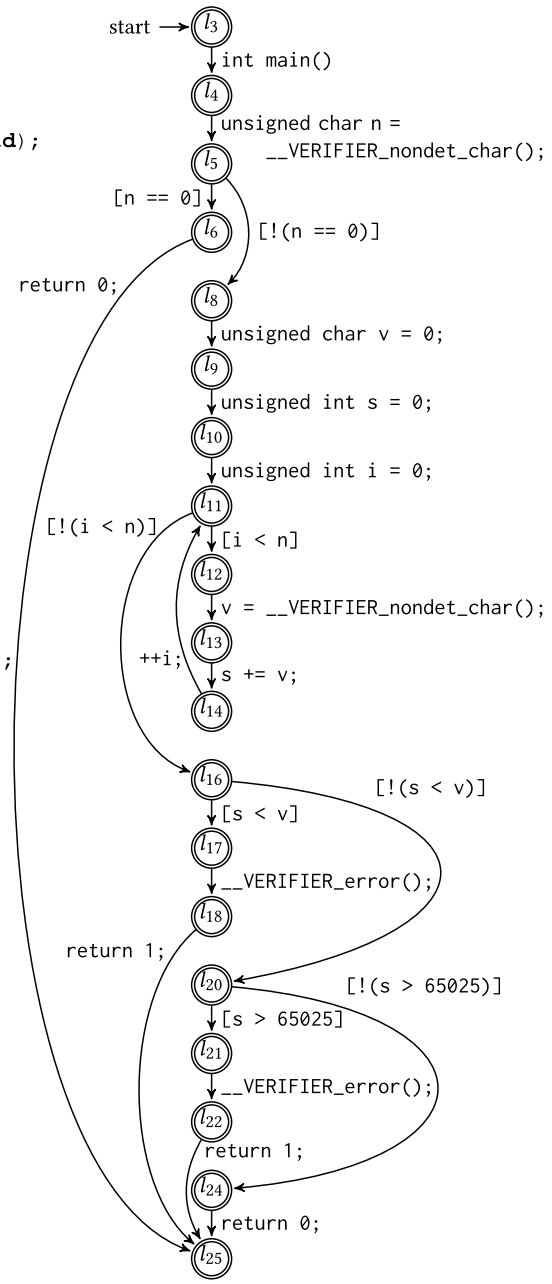
A sequence $\langle (l_0, op_1, l_1), \ldots \rangle$ of consecutive edges from G is called *program path* if it starts at the initial location, i.e., $l_0 = l_{init}$. A *test vector* [19] specifies the input values to a program. A program path is called *feasible*, if a test vector exists for which this program path is executed, otherwise the program path is called *infeasible*. A *concrete program path* is a feasible program path with variable assignments from a test vector attached to the locations along the path. An *error path* is a program path that violates a given specification.[2]

Example 1 (Program and Control-flow Automaton). Figure 1 shows the source code (Figure 1(a)) of an exampleC program that computes the sum of a number of input values and the corresponding CFA (Figure 1(b)). Location $l_{init} = l_3$ is the initial location of this program. The program contains four variables: `n` is the number of values to sum up, `v` is used to hold input values for the computation, `s` is the aggregation variable for the sum, and `i` is a loop counter. The type of the variables `n` and `v` is `unsigned char`, the type of the variables `s` and `i` is `unsigned int`, and for all our examples we will assume a data-type model where the type `unsigned char` has a bit width of 8, such that the values of this type range between `0` and `255`, and the type `unsigned int` has a bit width of 32, such that the values of this type range between `0` and `4294967295`. The CFA starts with the function entry in line 3, modeled by the edge from l_3 to l_4. The next CFA edge (l_4 to l_5) shows that the number of values to sum up, `n`, is initialized via the input function `__VERIFIER_nondet_char(void)` in line 4. In line 5, the program checks the value of `n` and immediately terminates in line 6 if it is `0`, which indicates that there are no values to be summed up. This check is modeled in the CFA by the branching at l_5, which goes from l_5 over l_6 to l_{25} in the early-return case and from l_5 to l_8 in the other case. In lines 8–10, the remaining variables `v`, `s`, and `i` are all initialized to `0`, which corresponds to the CFA edges from l_8 to l_{11}. The loop that computes the sum of input values read via `__VERIFIER_nondet_char(void)` in lines 11–15 is modeled by the CFA nodes l_{11}, l_{12}, l_{13}, and l_{14}, with l_{11} being the loop head. After the loop, there are two assertions: In line 16, the program checks that the sum is not less than the last value added to it, which seems like a sensible requirement given that all added values are non-negative. If the check fails, the program calls the function `__VERIFIER_error(void)` in line 17 to indicate an error and terminates in line 18. This check is modeled in the CFA by the branching at l_{16}, which goes from l_{16} over l_{17} and l_{18} to l_{25} in the failing case and from l_{16} to l_{20} in the other case. In line 20, the

[2]Details of how we represent and use specifications can be found in Section 3.2.1.

```
1 extern void __VERIFIER_error(void);
2 extern unsigned char
↪ __VERIFIER_nondet_char(void);
3 int main() {
4   unsigned char n =
  ↪ __VERIFIER_nondet_char();
5   if (n == 0) {
6     return 0;
7   }
8   unsigned char v = 0;
9   unsigned int  s = 0;
10  unsigned int  i = 0;
11  while (i < n) {
12    v = __VERIFIER_nondet_char();
13    s += v;
14    ++i;
15  }
16  if (s < v) {
17    __VERIFIER_error();
18    return 1;
19  }
20  if (s > 65025) {
21    __VERIFIER_error();
22    return 1;
23  }
24  return 0;
25 }
```

(a) Program source code



(b) CFA

Fig. 1. Example C program linear-inequality-inv-a.c as source code (a) and as a CFA (b).

program checks that the sum is not greater than `65,025`, which is the product of the maximum value of `n` and therefore the maximum number of values that are added up, `255`, and the maximum value of each value being added up, which is also `255`. If the check fails, the program calls the function `__VERIFIER_error(void)` in line 21 to indicate an error and terminates in line 22. This

check is modeled in the CFA by the branching at l_{20}, which goes from l_{20} over l_{21} and l_{22} to l_{25} in the failing case and from l_{20} to l_{24} in the other case, where the program terminates with return code 0 (success).

2.2 Configurable Program Analysis

The concept of ***configurable program analysis*** (**CPA**) [30, 34] allows the separation of the definition of the abstract domain that is used for a program analysis from the analysis algorithm. A CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$ specifies an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge, and a stop operator stop, all of which configure the CPA algorithm and are explained in the following. The CPA algorithm can be used with any CPA and is an algorithm for reachability analysis. It is possible to combine a set of CPAs into a single, composite, CPA (see Section 2.2.2).

The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ is composed of a set C of concrete states where each concrete state $c \in C$ is a total function of type $X \to \mathbb{Z}$ (i.e., a concrete state is a mapping from program variables to integers), a semilattice $\mathcal{E} = (E, \sqsubseteq)$ over a set E of abstract states (i.e., abstract-domain) and a partial order $\sqsubseteq$ (the join $\top$ (least upper bound) of all elements and the join $\sqcup$ of two elements are unique, but a unique element $\bot$ (greatest lower bound) is not required), and a concretization function $[\![\cdot]\!]$ that maps each abstract state to the set of concrete states represented by that abstract state. The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ specifies for each abstract state $e \in E$ and control-flow edge $g \in G$ its abstract successor states, i.e., the abstract states that overapproximate the concrete successor states of all concrete states represented by e via the control-flow edge g. The merge operator $\text{merge} : E \times E \to E$ defines if and how to merge two abstract states $e, e' \in E$ when control flow meets. The stop operator $\text{stop} : E \times 2^E \to \mathbb{B}$ decides for an abstract state $e \in E$ and a given set $R \subseteq E$ of abstract states whether e is covered by R. The level of abstraction the analysis operates on can be configured by choosing the operators merge and stop appropriately. Two common choices for these operators are $\text{merge}^{sep}(e, e') = e'$, which does not combine abstract states, and $\text{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$, which checks whether the given abstract state e is less than or equal to ("covered by") any abstract state e' from R according to the semilattice $\mathcal{E}$ to determine coverage.

2.2.1 CPA Algorithm. The CPA algorithm takes a CPA and an initial abstract state as input (Algorithm 1). Essentially, the algorithm performs a classic fixed-point iteration by computing successor states of reached abstract states until the set waitlist of unprocessed states is empty (i.e., until all reachable abstract states have been completely processed) and returns the set reached of reachable abstract states. In each major iteration, the algorithm takes one abstract state e from the waitlist, computes all its abstract successors, and processes each of them separately: For each successor abstract state e' the algorithm uses the operator merge to check if an already explored abstract state e'' with which the successor abstract state e' should be merged exists in the set reached of reached states (e.g., at meet points where the control flow of different paths meets after completed branching). If the operator merge decides that the two abstract states should be combined, then the existing abstract state e'' is substituted by the new, merged abstract state e_{new} in both sets reached and waitlist. The stop operator implements the detection of a fixed point. The CPA algorithm uses it to check if the new abstract state e' is already covered by an existing abstract state in the set reached, and only if the result is negative it inserts the new abstract state e' into the work sets waitlist and reached, i.e., only if this is necessary to explore this abstract state further.

2.2.2 Composite CPA. A *Composite CPA* [34] can be used to combine a set of CPAs into a single, composite, CPA. An abstract state of the Composite CPA is a tuple composed of one component abstract state for each component CPA and the operators merge and stop are defined to delegate to

ALGORITHM 1: CPA($\mathbb{D}$, e_{init}), taken from [34]

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
where E is the set of elements of the semilattice of D,
and an initial abstract state $e_{init} \in E$,
Output: a set of reachable abstract states
Variables: two sets reached and waitlist of elements of E

```
1:  reached := {e_init}
2:  waitlist := {e_init}
3:  while waitlist ≠ ∅ do
4:      pop e from waitlist
5:      for all e' with e⇝e' do
6:          for all e'' ∈ reached do
7:              e_new := merge(e', e'')
8:              if e_new ≠ e'' then
9:                  waitlist := (waitlist ∪ {e_new}) \ {e''}
10:                 reached := (reached ∪ {e_new}) \ {e''}
11:         if not stop(e', reached) then
12:             waitlist := waitlist ∪ {e'}
13:             reached := reached ∪ {e'}
14: return reached
```

the component CPAs' respective operators, such that the merge operator combines abstract states according to how the components' merge operators combine the component abstract states, and the operator stop only returns *true* if all components agree that their component abstract states are already covered by their respective existing component abstract states in the set reached.

Consequently, such a combination of CPAs automatically causes all used CPAs to implicitly cooperate on discarding infeasible program paths during the program analysis, because a composite abstract successor state for a given composite abstract state is only produced if all component CPAs produce a component abstract successor state for their respective component abstract state. Thus, if one component CPA is able to prove that a specific program path is infeasible, that path no longer needs to be considered by any other component CPA either, and the composite analysis will only find program paths that all component CPAs consider feasible. Note that no explicit communication between the component CPAs is required and that the component CPAs do not even need to know their sibling components exist to achieve this effect. If desired, however, such an explicit information exchange is possible via the strengthen operator $\downarrow$ [34] of the Composite CPA, which can be used to further improve precision.

2.2.3 Location CPA. A very basic CPA that we will use for all our analyses is the *Location CPA* $\mathbb{L}$, which tracks the program counter. The Location CPA uses a flat lattice overall program locations and the operators merge^{sep} and stop^{sep}. Using this component, we are able to effectively separate the concern of tracking program locations from other concerns[3] and do not need to re-implement this feature for every analysis.

3 CONCEPTS

The purpose of verification witnesses is to represent information about verification results in such a manner that it is machine-readable, reproducible, and exchangeable between verification tools.

[3]Specifically, the semantics of the program is analyzed and tracked in other CPAs, not in the Location CPA.

There are two types of verification witnesses: Violation witnesses, which represent error paths that violate a specification, and correctness witnesses, which represent the artifacts of a proof that a program satisfies a specification. In this section, we present the basic concepts for verification witnesses whereas the specifics of violation witnesses and correctness witnesses will be discussed in a later section on design and implementation (Section 4).

3.1 Protocol Automata

We define *protocol automata* [25, 30, 45], which we instantiate later to *witness automata* to represent witnesses, and to *observer automata* to represent specifications.

A *protocol automaton* $A = (Q, \Sigma, \delta, q_{init}, F)$ for a CFA (L, l_{init}, G) is a nondeterministic finite automaton and its components are defined as follows:

(1) The set $Q \subseteq \Gamma \times \Phi$ is a finite set of control states, where each control state $q \in Q$ has a unique name γ from a set Γ of names, which can be used to uniquely identify a control state q within Q, and an invariant φ from the set Φ of predicates of a given theory.
(2) The set $\Sigma \subseteq 2^G \times \Phi$ is the alphabet, in which each symbol $\sigma \in \Sigma$ is a pair (S, ψ) that comprises a finite set $S \subseteq G$ of CFA edges and a state condition $\psi \in \Phi$.
(3) The set $\delta \subseteq Q \times \Sigma \times Q$ contains the transitions between control states, where each transition is a triple (q, σ, q') with a source state $q \in Q$, a target state $q' \in Q$, and a guard $\sigma = (S, \psi) \in \Sigma$ comprising a *source-code guard* S (see Example 3), which restricts a transition to the specific set $S \subseteq G$ of CFA edges, and a *state-space guard* $\psi \in \Phi$, which restricts the state space to be considered by an analysis that consumes the protocol automaton. We also write $q \xrightarrow{\sigma} q'$ for $(q, \sigma, q') \in \delta$.
(4) The control state $q_{init} \in Q$ is the initial control state of the automaton.
(5) The subset $F \subseteq Q$ contains the accepting control states.

We define a number of properties for protocol automata:

Sink Control State. A state $q \in Q$ is called a *sink control state*, if $q \notin F$ and $\nexists q \xrightarrow{\sigma} q' \in \delta$, i.e., sink states are not accepting and do not have outgoing transitions.

Stutter-enabled State. A state $q \in Q$ is called *stutter-enabled*, if there is a special self-transition $q \xrightarrow{o/w} q \in \delta$ (with the special guard symbol "*o/w*" short for "otherwise"), which is defined as follows: Let $\delta_{q,other}$ be the set of all outgoing transitions of q except those with the guard *o/w*, i.e., $\delta_{q,other} = \{q \xrightarrow{\sigma} q' | \sigma \neq o/w\}$. The transition $q \xrightarrow{o/w} q$ is a self-transition where the state-space guard ψ is *true* and the source-code guard S matches the set of all CFA edges that are either (a) not matched by the source-code guard of any other outgoing transition of q or (b) are matched by the source-code guard of some other outgoing transition of q that also matches a successor CFA edge. Thus, a stutter-enabled state ensures that for every CFA edge, there is always at least one outgoing transition of the state where the source-code guard matches the transition, which is a requirement of our witness automata (see Section 3.1.2). Moreover, the mechanism can be used to support nondeterminism, because if there is a transition with a source-code guard that ambiguously matches two (or, transitively, more) consecutive CFA edges, there is also a matching self-transition that does not impose a state-space guard. More formally, the transition $q \xrightarrow{o/w} q$ is equivalent to a transition $q \xrightarrow{(S_{q,stutter}, true)} q$ with $S_{q,stutter} = (G \setminus S_{q,other}) \cup S_{q,ambig}$, where $S_{q,other}$ is the set of all CFA edges that are matched by the source-code guard S' of any other outgoing transition of q, i.e., $S_{q,other} = \bigcup \{S' | q \xrightarrow{(S',\cdot)} q' \in \delta_{q,other}\}$, and $S_{q,ambig}$ is the set of those CFA edges matched by any other outgoing transition of q that has a successor CFA edge that is matched by the same transition, i.e., $S_{q,ambig} = \{s \in G | \exists q \xrightarrow{(S,\cdot)} q' \in \delta_{q,other},\ s, s' \in S : s = (\cdot, \cdot, l_b) \wedge\ s' = (l_b, \cdot, \cdot)\}$.

3.1.1 Control-flow Automata. A *control-flow automaton* can be seen as a special kind of protocol automaton for which

- all states are accepting (i.e., $F = Q$),
- no sink states exist,
- all invariants are *true*, formally: $\forall(\cdot, \varphi) \in Q : \varphi = \mathit{true}$, and
- the transition labels contain only a singleton of one control-flow edge and all guards are *true*, formally: $\forall(S, \psi) \in \Sigma : |S| = 1, \psi = \mathit{true}$.

As a consequence, control-flow automata are non-restricting protocol automata, because they cannot be used to restrict state-space exploration when used in a protocol analysis as defined in Section 3.3.

3.1.2 Witness Automata. A *witness automaton* is a protocol automaton with the requirement that there must be an outgoing transition from every non-sink control state for every CFA edge, such that every program path can be simulated in the automaton unless the simulation is explicitly terminated by a sinking state, i.e., for every non-sink control state $q \in Q$ and for every CFA edge $g \in G$, *some* transition $q \xrightarrow{(S,\cdot)} q' \in \delta$ must exist with $g \in S$. To fulfill the requirement above and as a mechanism to allow ambiguity (because in practice, it is not always convenient or even feasible for the producer of a protocol automaton to precisely describe the source-code guard of a transition), we require that every non-sink control state $q \in Q$ is *stutter-enabled*, i.e., $\forall q \in \{r \in Q \mid r \in F \vee \exists r \xrightarrow{\sigma} r' \in \delta\} : q \xrightarrow{o/w} q \in \delta$. In the exchange format for verification witnesses (see Section 5), these *o/w*-transitions are not written explicitly, because they exist by definition.

Example 2 (Handling Ambiguity). Consider a C program with the following statements:

```
1  int c = 0;
2  int x = 1; ++x; ++x;
3  if (c == 0) { __VERIFIER_error(); }
```

Assume that there is a verifier that knows that the assumption $x = 3$ holds after line 2 and wants to produce a protocol-automaton transition to convey this information. The best way to precisely convey this information would be to use a source-code guard that matches only the CFA edge for the last statement in line 2. If, however, the program representation used internally by the verifier only retains the line numbers of statements, the verifier is only able to specify that the assumption $x = 3$ holds after some statement in line 2. If the protocol automaton would then (deterministically) enforce the state-space restriction $x = 3$ after the first statement (first match) in line 2, the restricted state space would be empty due to the contradiction with the fact that $x = 1$. However, because of the requirement that every control state of the protocol automaton must handle such ambiguous matches nondeterministically, the automaton can "wait" until the last statement in line 2 to apply the state-space guard. The downside of this approach is that the non-determinism inflates the search space through the automaton. This downside can be mitigated by strong state-space guards that lead to contradictions early on wrong paths through the automaton.

When a consumer of a protocol automaton, e.g., a witness-based result validator, uses the automaton to guide its exploration of the state space, the exploration can be restricted either by restricting the state space using state-space guards at transitions or by a transition to a sink control state (that, by definition, has no outgoing transition, and thus the path exploration ends).

3.1.3 Observer Automata. An *observer automaton* (also called "monitor automaton" [118]) for a given CFA $C = (L, l_{init}, G)$ is a protocol automaton $(Q, \Sigma, \delta, q_{init}, F)$ that satisfies the following conditions:

(1) there are no sink control states,
(2) all invariants are *true*, formally: $\forall(\cdot, \varphi) \in Q : \varphi = \mathit{true}$, and
(3) for every control state $q \in Q \setminus F$ and every CFA edge g of G, the disjunction $\bigvee \left\{\psi \,\middle|\, \exists\, q \xrightarrow{(S,\psi)} \cdot \in \delta : g \in S\right\}$ of all state-space guards for q and g evaluates to *true*, i.e., state-space guards may be used to partition the state space of the program, but not to restrict it. There must be at least one transition for every q and g to satisfy this condition.

Note that it might be useful to have several transitions in the observer automaton for one CFA edge. To ensure that the disjunction of the guards evaluates to *true*, users can use a SPLIT transition (syntactic sugar, see Section 5.4 of [30]).

3.1.4 Abstract Reachability Graphs. An ***abstract reachability graph*** (**ARG**) [31] can be seen as a protocol automaton where the set of states is given by the set reached of reachable abstract states that were discovered by a reachability analysis. A transition $e \xrightarrow{\sigma} e'$ exists if e' is either an abstract successor state of e or e' is the result of merging an abstract successor of e with some other abstract state(s). For an abstract state $e = (\cdot, \varphi)$, the invariant describes the set of concrete states that the abstract state represents. The transition label $\sigma = (S, \psi)$ consists of a guard that is always true ($\psi = \mathit{true}$) and a set of control-flow edges S that is either a singleton $S = \{g\}$ that contains the control-flow edge $g \in G$ that was taken from e to e' or the empty set $S = \{\}$ that indicates a coverage relation if $e \sqsubseteq e'$. An *abstract path* through an ARG is a sequence $\langle e_0, \ldots, e_n \rangle$ of abstract states such that every pair (e_i, e_{i+1}) with $i \in \{0, \ldots, n-1\}$ is an edge in the ARG. ARGs are used in software verification to represent correctness proofs (if the invariants in the abstract states are inductive) and violation proofs (if it contains an abstract path that represents an error path).

3.1.5 Floyd–Hoare Automata. A *Floyd–Hoare automaton* [84] is an observer automaton with the following constraints:

— the initial state has the invariant *true*,
— all state-space guards are *true*,
— for each transition $(\cdot, \varphi_q) \xrightarrow{(S,\mathit{true})} (\cdot, \varphi_{q'}) \in \delta$ and each operation $op \in \{op \mid \exists(\cdot, op, \cdot) \in S\}$, the triple $\{\varphi_q\}\ op\ \{\varphi_{q'}\}$ is a valid Hoare triple, and
— each accepting state has the invariant *false*.

Hence, a Floyd–Hoare automaton accepts only sequences of operations that are infeasible.[4] Floyd–Hoare automata are used in software verification to represent correctness proofs.

3.1.6 Run. Let $p = \langle (l_0, op_1, l_1), \ldots \rangle$ be a path with $l_0 = l_{init}$, i.e., a program path, and let $\widehat{p}$ be a concrete program path for p. A *run* for this concrete program path $\widehat{p}$, and thus, also for p, is a simulation sequence $\langle q_{init} \xrightarrow{\sigma_1} q_1, \ldots \rangle$ such that for all program locations $l_i \neq l_0$ of $\widehat{p}$, the ith CFA edge (l_{i-1}, op_i, l_i) of $\widehat{p}$ is matched by the ith transition $q_{i-1} \xrightarrow{\sigma_i} q_i$ of the simulation sequence, with $\sigma_i = (S, \psi)$ and $(l_{i-1}, op_i, l_i) \in S$, and all variable assignments that are attached to the ith program location of $\widehat{p}$ satisfy ψ.

3.1.7 Acceptance. Protocol automata provide flexibility regarding the acceptance criterion in that they allow a choice: If the goal is to accept *finite* runs (e.g., for the witness-based validation of verification results for reachability problems, as in the examples in this article), a protocol automaton A can be defined to accept the run $\langle \ldots, q_{n-1} \xrightarrow{\sigma_n} q_n \rangle$ if $q_n \in F$. If the goal is to accept *infinite* runs (e.g., for the witness-based validation of verification results for termination problems), we can define the protocol automaton A as a Büchi automaton that accepts an infinite run ρ if there

[4]Note that the invariants of such a Floyd–Hoare automaton can be computed using Craig interpolation [31, 66, 108, 109].

exists a control state $q \in F$ that occurs infinitely often in ρ. We say A accepts a program path p if there exists an accepting run of A for p. The projection of an accepted finite run to the sequence $\langle \sigma_1, \ldots, \sigma_n \rangle$ of its alphabet symbols is called an *accepted word*, as is the projection of an accepted infinite run to the sequence $\langle \sigma_1, \ldots \rangle$ of its alphabet symbols. The set of all accepted words of A defines the *language* $\mathcal{L}(A)$.

3.1.8 Graphical Representation. In this article, we will give several graphical examples of protocol automata. We draw them as graphs where the control states are circular nodes. We mark the initial control state with an incoming edge that has no source node and is labeled "start". We label each control state with its name and present its invariant as a boolean expression in a green-colored box next to the state, except if every control state in the automaton has the invariant *true*, in which case we omit the invariants from the figure. We mark sink states by coloring them blue. We mark accepting control states with a double border. If a control state is intended to represent a specification violation, we color it red. We draw the transitions as edges and label each of them with the following syntax: The label is split into two parts by a colon. The first part (i.e., the part before the colon) corresponds to the source-code guard, which is given as a comma-separated list of tokens that define a set of matched CFA edges conjunctively. A numerical token describes a line number and restricts the set of matched CFA edges to edges that represent an operation on that source-code line. The second part of the edge label (i.e., the part after the colon) corresponds to the state-space guard and is given as a boolean expression, except if it is *true*, in which case we omit it.

Example 3 (Source-code Guards). In the CFA in Figure 1(b), the token "`4`" matches the CFA edge from l_4 to l_5, whereas the token "`5`" matches the edges from l_5 to l_6 and from l_5 to l_8. A token "`enterFunction(`f`)`" restricts the set of matched edges to edges that represent a function-call operation to function f. In Figure 1(b), the token "`enterFunction(main)`" matches the edge l_3 to l_4, and the token "`enterFunction(__VERIFIER_error)`" matches the edges l_{17} to l_{18} and l_{21} to l_{22}. The tokens "`then`" and "`else`" restrict the set of edges to edges that correspond to the positive case of a conditional branching ("`then`", where the condition evaluates to *true*), or the negative case of the branching ("`else`", where the condition evaluates to *false*). For example, such a branching may be a loop condition or an if statement. In Figure 1(b), the edges l_5 to l_6, l_{11} to l_{12}, l_{16} to l_{17}, and l_{20} to l_{21} are matched by the token "`then`", whereas the edges l_5 to l_8, l_{11} to l_{16}, l_{16} to l_{20}, and l_{20} to l_{24} are matched by the token "`else`". A token "`enterLoopHead`" restricts the set of edges to those that precede a loop head. In Figure 1(b), the edges l_{10} to l_{11} and l_{14} to l_{11} are matched by the token "`enterLoopHead`".

3.2 Automata Representations

The various kinds of protocol automata are used to represent specifications, violation witnesses, and correctness witnesses. Table 1 gives an overview of the kinds of automata that are used conceptually as representation, and the specific characteristics (cf. also [45]).

3.2.1 Specifications Represented by Observer Automata. Using observer automata to model formal specifications is an established concept [6, 20, 38, 118, 120]; consequently, we also use an observer automaton to model safety specifications. Separating the specification from the implementation follows the best-practice of separation of concerns. As a result, we can check a given programs against different specifications without changing the source code, and we can also use a given specification to check different programs. We call a given pair of program and specification a *verification task*. Note that for practical reasons we configure the observer automata such that they accept paths that *violate* the specification (cf. [45]).

Table 1. Mapping Artifacts Related to Software Analysis to Their Representing Types of Automata and to Examples of Analyses That Use These Artifacts

Artifact	Type of Automaton	Type of Analysis	all states accepting	no sink states	all invariants true	non-restricting
Program	CFA	location analysis (cf. Section 2.2.3)	✓	✓	✓	✓
Specification	observer automaton	observer analysis (cf. Section 3.3)		✓	✓	✓
Violation Witness	witness automaton	protocol analysis (cf. Section 3.3)			✓	
Correctness Witness	witness automaton	observer analysis (cf. Section 3.3)	✓	✓		✓
Proof	ARG	composite analysis (cf. Section 2.2.2)	✓	✓		✓
Proof	Floyd–Hoare automaton	Floyd–Hoare analysis (cf. Section 4.1.2)		✓		✓

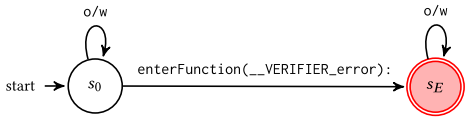


Fig. 2. Specification that forbids calls to the function `__VERIFIER_error(void)`, represented as observer automaton that accepts all program paths that enter control state s_E, i.e., violate the specification.

Example 4 (Specification). Figure 2 shows an example of a specification represented by an observer automaton that forbids calls to a function `__VERIFIER_error(void)`, i.e., this specification is violated by a program if there is a feasible program path that contains a call to the function `__VERIFIER_error(void)`. The program represented by the CFA in Figure 1(b) does not violate this specification because both CFA nodes l_{18} and l_{22} are not reachable.

3.2.2 Violation Witnesses Represented by Witness Automata. A *violation-witness automaton* is a witness automaton, i.e., a protocol automaton that represents a witness, in this case more specifically, a violation witness. Violation-witness automata use the state-space restricting features of protocol-automata to guide the exploration towards the specification violation. In a violation-witness automaton, the set of accepting (violation) control states contains only those states that correspond to violating program states detected by the producing verifier.

A *violation-witness automaton* is a witness automaton for which

— all invariants are *true*, formally: $\forall(\cdot, \varphi) \in Q : \varphi = \mathit{true}$.

3.2.3 Correctness Witnesses Represented by Witness Automata. A *correctness-witness automaton* is a witness automaton, i.e., a protocol automaton that represents a witness, in this case more specifically, a correctness witness. Correctness-witness automata do not use the state-space restricting features of protocol-automata: While a violation-witness automaton may restrict the successor states to those successor states that lead the exploration to the specification violation, a correctness-witness automaton has abstract successor states for all concrete successor states. The correctness-witness automaton annotates each abstract program state e with an invariant φ, i.e., a predicate that holds at e on every program path that passes e. In a correctness-witness automaton, the set of accepting control states is equivalent to the (whole) set of control states.

A *correctness-witness automaton* is a witness automaton for which

- all states are accepting (i.e., $F = Q$),
- there are no sink control states, and
- for every control state $q \in Q$ and every CFA edge g of G, the disjunction $\bigvee \left\{ \psi \mid \exists\, q \xrightarrow{(S,\psi)} \cdot \in \delta : g \in S \right\}$ of all state-space guards for q and g evaluates to *true*, i.e., state-space guards may be used to partition the state space of the program, but not to restrict it.

3.3 Automaton CPA: A Configurable Program Analysis for Protocol Automata

A *protocol analysis* is an *Automaton CPA* $\mathbb{O} = (D_{\mathbb{O}}, \rightsquigarrow_{\mathbb{O}}, \text{merge}_{\mathbb{O}}, \text{stop}_{\mathbb{O}})$ for a protocol automaton $A = (Q, \Sigma, \delta, q_{init}, F, B)$. An Automaton CPA is a CPA (cf. Section 2.2) that tracks the control state of A. The Automaton CPA comprises the following components, for a given CFA $C = (L, l_{init}, G)$ (cf. [30]):

(1) $D_{\mathbb{O}} = (\mathcal{C}, \mathcal{Q}, [\![\cdot]\!])$ is an abstract domain comprising the set $\mathcal{C}$ of concrete states, the semi-lattice $\mathcal{Q}$ over abstract data states, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{Q} = (Z, \sqsubseteq)$ consists of the set Z of abstract data states and a partial order $\sqsubseteq$ (the join $\sqcup$ and the top element $\top_{\mathcal{Q}} = (\top, \mathit{true})$ are unique). An abstract state of $Z = (Q \cup \{\top_{\mathcal{Q}}\})$ is either a control state from Q (a pair of a name from Γ and an invariant from Φ, the set of predicates of a given theory) or the special lattice element $\top_{\mathcal{Q}}$. The definition of the partial order $\sqsubseteq$ is that $(\gamma, \varphi) \sqsubseteq (\gamma', \varphi')$ if $(\gamma' = \top \vee \gamma = \gamma') \wedge \varphi \Rightarrow \varphi'$. The join operator $\sqcup$ is defined as the least upper bound of two abstract data states. The top element $\top_{\mathcal{Q}}$ is the least upper bound of all abstract data states, i.e., $\forall (\gamma, \varphi) \in Z : (\gamma, \varphi) \sqsubseteq \top_{\mathcal{Q}}$. The concretization function $[\![\cdot]\!] : Z \to 2^C$ assigns to each abstract data state (γ, φ) the corresponding set $[\![\varphi]\!]$ of concrete states.

(2) $\rightsquigarrow_{\mathbb{O}} \subseteq Z \times G \times Z$ is the transfer relation. A transfer $(\gamma, \cdot) \overset{g}{\rightsquigarrow}_{\mathbb{O}} (\gamma', \varphi')$ exists if the protocol automaton A has a matching transition $(\gamma, \cdot) \xrightarrow{(S,\psi')} (\gamma', \varphi')$ with $\varphi' = \psi'$ and $g \in S$. Because the condition ψ' of the protocol-automaton transition is stored in the successor abstract data state, it is accessible to other component analyses via the composite strengthening operator (cf. Section 2.2.2) and can be used by them to strengthen their own successor abstract data states.

(3) Only elements with the same control-state name are combined by the merge operator: $\text{merge}_{\mathbb{O}}((\gamma, \varphi), (\gamma', \varphi')) = \begin{cases} (\gamma', \varphi \vee \varphi') & \text{if } \gamma = \gamma' \\ (\gamma', \varphi') & \text{otherwise} \end{cases}$

(4) $\text{stop}_{\mathbb{O}}((\gamma, \varphi), R)$ is the termination check. It terminates the state-space exploration of the current path (i.e., it returns *true*) if the abstract data state (γ, φ) is covered by an existing abstract data state in R: $\text{stop}_{\mathbb{O}}((\gamma, \varphi), R) = \exists (\gamma, \varphi') \in R : \varphi \Rightarrow \varphi'$

Witness Analysis. A *witness analysis* is an Automaton CPA for a witness automaton.

Observer Analysis. An *observer analysis* is an Automaton CPA for an observer automaton, i.e., an observer analysis only "observes" (or "monitors") the paths of the analyzed program, but it does not restrict the exploration performed by the program analysis. One use case for such an observer analysis is to observe whether an analyzed program path violates the specification, i.e., to determine whether it is an error path. The accepted program paths are those that violate the specification. An observer CPA can also be used to split abstract paths and observe them separately.

3.4 Constructing Witness Automata from Proofs

A verification tool can produce witnesses by transforming the desired paths of the constructed proof (which is available from most types of program analysis, including the configurable program analysis described in Section 2.2) into a witness automaton.

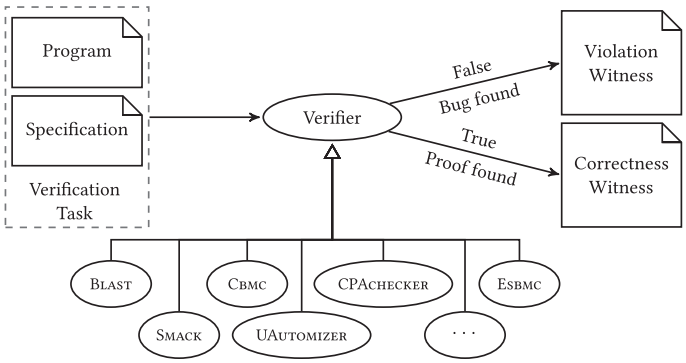


Fig. 3. Software verifiers produce witnesses.

3.4.1 Witness Automata from ARGs. The nodes of the ARG become control states in the witness automaton, with the root node of the ARG as the initial control state q_{init} of the witness automaton. The edges of the ARG become transitions in the witness automaton. Edges that leave the desired paths of the ARG to become transitions to a sinking state, i.e., a state with no outgoing transitions. To each transition, the verifier should add a source-code guard that describes the CFA edge represented by the corresponding ARG edge as precisely as possible. Constraints on variable values at the target state of an ARG edge may be encoded as state-space guards of the corresponding transition (for violation witnesses), or as a control-state invariants of the corresponding control state (for correctness witnesses). After producing the witness automaton from the ARG, several minimizations can be performed. For example, control states that are connected only by transitions without any guards and have the same control-state invariant can be merged, because they do not convey any useful information.

3.4.2 Witness Automata from Floyd–Hoare Automata. Witness automata can also be derived from Floyd–Hoare automata using a similar construction.

3.5 Application Scenarios

In the following, we describe scenarios where the concept of verification witnesses is applied.

3.5.1 Verification with Witnesses. Good practice requires a verifier, whenever it reaches a conclusion regarding a given verification task, to produce a verification witness that provides information about the verification result. The purpose of the verification witness is to document the verification result and to make valuable verification artifacts available for reuse instead of leaving unused the effort the verifier has already spent on them. The primary use case we discuss in this article is witness-based result validation. Another use case is the visualization of verification results [22, 114]. Figure 3 illustrates the process of verification with witnesses, and it also shows one key feature of the concept of having a common representation of verification results: there is no risk of technology lock-in, because the verifiers are interchangeable according to the needs of the user.

3.5.2 Witness-based Result Validation. A witness-based result validator can independently reestablish a verification result of a verifier using the guidance of a verification witness. We describe a program analysis for this purpose using the CPA concept by configuring a Composite CPA with the following components: One component is an Automaton CPA that performs a protocol analysis (cf. Section 3.3) for a witness automaton (which we also call witness analysis, because

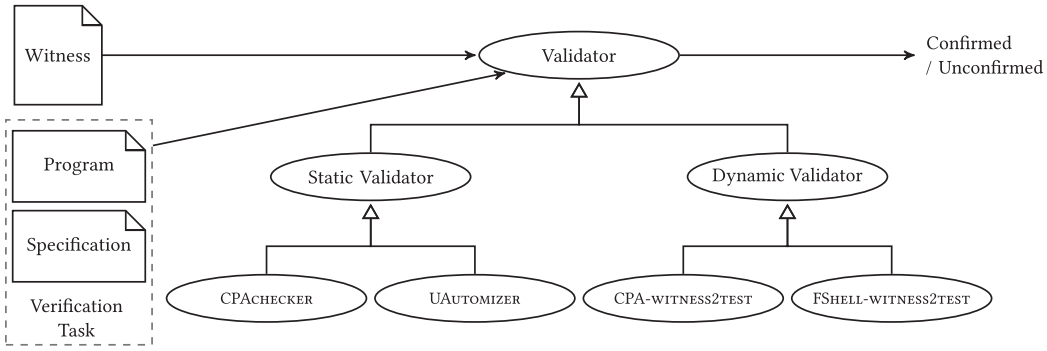


Fig. 4. Witness-based result validation.

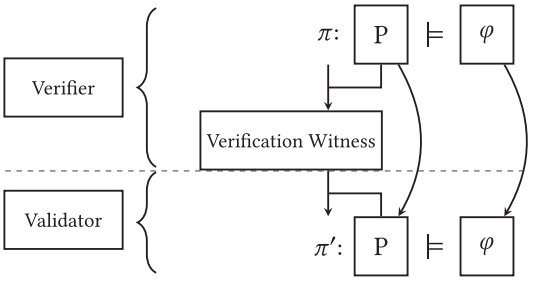


Fig. 5. In witness-based result validation, a verification witness is produced by a verifier that is able to find a proof π that proves or disproves $P \models \varphi$. The verification witness carries information that may guide the validator to find its own proof π' that also proves or disproves $P \models \varphi$.

it simulates the witness automaton). One of the other component CPAs is an Automaton CPA that performs an observer analysis (cf. Section 3.3) and encodes the specification, which is represented by an observer automaton (cf. Section 3.1.3), i.e., which only observes but does not restrict how the program's state space is explored by the program analysis. The composed program analysis only considers specification violations signaled by this CPA if the CPA that simulates the witness automaton agrees, i.e., if both the specification automaton *and* the witness automaton accept the corresponding run. Another component of CPA is the Location CPA, which is used to track the program counter, i.e., the location in the CFA, for the analysis. Further components can be added to the composition, for example, to track information about the values of program variables. These components can then use the operator $\downarrow$ of the Composite CPA to compute the intersection of their component abstract states with the state-space guards from the witness automaton to achieve a restricted, more precise state space, or they can check the validity of the state invariants of the witness automaton, and, if successful, use these invariants as proof lemmas. Figure 4 illustrates the process of witness-based result validation and shows four existing implementations of validators. A verification result is confirmed by a witness-based result validator if the validator is able to re-establish the verification result.

We illustrate witness-based result validation with Figure 5: First, the verifier receives the verification task $P \models \varphi$ as input and constructs a proof π for proving or disproving the statement. In the former case, the verifier produces a correctness witness (which contains invariants) and in the latter case, the verifier produces a violation witness (which contains an error path). Second, a validator receives the same verification task $P \models \varphi$ and the witness as input but constructs a

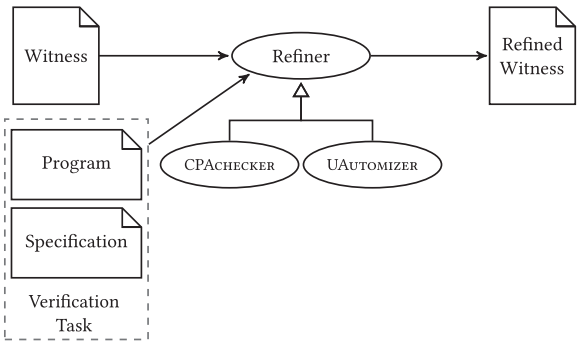


Fig. 6. Witness refinement.

new proof π' which might be different from the verifier's proof π. However, the validator is allowed to look into the verification witness in order to obtain information to support the proving process. If it receives a correctness witness as input, then it tries to use the invariants (i.e., using inductiveness checks); if it receives a violation witness as input, then it tries to use the error path (i.e., using a simulation to check feasibility).

3.5.3 Witness Refinement. Witness refinement is the iterative process of improving witnesses, by augmenting the verification artifacts represented by an input witness using new, potentially more detailed, information computed by a witness-based result validator. This process combines the concept of witness-producing verification with the concept of witness-based result validation. Figure 6 illustrates the process of witness refinement and shows the two existing implementations of witness refinement. We call a tool that is able to perform witness refinement a witness refiner. By using the common exchange format for verification witness, witness refinement can be applied using any desired sequence of witness refiners that are available to a user.

3.5.4 Witnesses for Concurrent Systems. For the ease of presentation, we restrict our descriptions to non-concurrent systems. The presented concepts, however, are also applicable to concurrent systems [29]: State-space guards and state invariants can be specified for specific threads. In a context-bounded view of a concurrent system (which is sufficient to describe a violation of a reachability property, for example,), we can consider a *concurrent CFA* as a product of the original CFA and the (maximum) number of concurrent threads, and describe a schedule by using source-code guards to specify the set of CFA edges in the concurrent CFA as a combination of a set of CFA edges from the original CFA and the set of potentially active threads.

3.5.5 Witnesses for Termination. While we use a reachability specification for our running example, the presented concepts are also applicable to the termination. Instead of using the acceptance criterion for finite runs, we can treat the witness automata as Büchi automata and use the acceptance criterion for infinite runs, as described in Section 3.1.7.

4 DESIGN AND IMPLEMENTATION

In the following, we will explain how we use protocol automata to represent verification results and how we can validate and refine these verification results by applying witness-based verification-result validators.

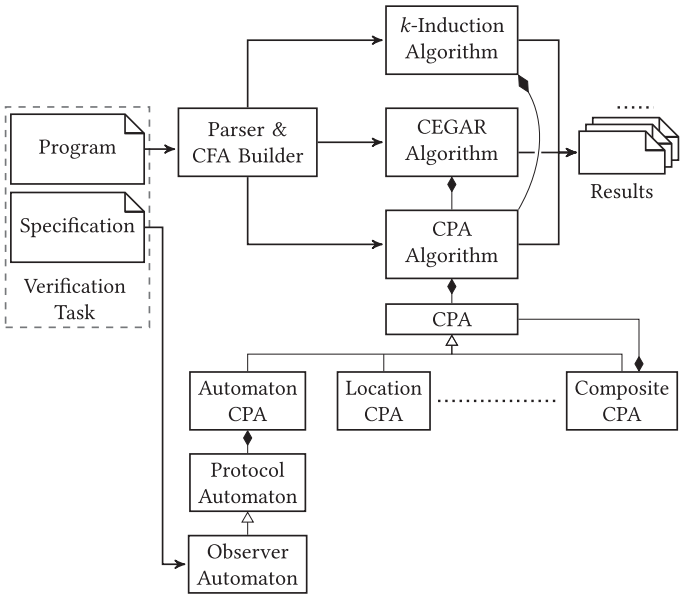


Fig. 7. The architecture of CPAchecker.

4.1 Verifiers

Over the last decades, a multitude of automated software verifiers has been developed. We now give a brief introduction to two verification tools that we will use to generate witnesses for our evaluation and that three of the four violation-witness-based result validators and both of the correctness-witness-based result validators we present in the following sections are based on.

4.1.1 CPAchecker. The configurable software-verification framework CPAchecker [38] is based on configurable program analysis [30, 34] and supports many different verification approaches, such as predicate abstraction [21, 39, 77], lazy abstraction with interpolants [46, 109], *k*-induction [27, 69], bounded model checking [49], explicit-value analysis [42], and symbolic execution [40]. CPAchecker won the category *Overall* of the competition on software verification (SV-COMP) seven times from 2012–2021.[5]

Architecture. Its architecture is designed to explicitly reflect the concepts of configurable program analysis (cf. Section 2.2) in its components, as visualized by Figure 7: The left side of the figure shows the input, which is a verification task that consists of a program and a specification. The source code of a verification task is parsed and converted into a CFA (cf. Section 2.1), the specification is parsed and converted into an observer automaton (cf. Section 3.2.1). Then, the desired algorithm is run on the verification task. After the algorithm completes, the computed results, for example, the verification outcome, are delivered. The core algorithm of the CPAchecker framework is the CPA algorithm (data-flow analysis/abstract interpretation [65, 101, 113]). As examples of CPAs implemented in CPAchecker, the figure shows the Location CPA and the Composite CPA, which were already introduced in Section 2.2, as well as the Automaton CPA, which is described in Section 3.3. The dotted line symbolizes that more CPAs that are not discussed in this article are available, for example, a CPA for predicate analysis [28] or a CPA for explicit-state model

[5] https://cpachecker.sosy-lab.org/achieve.php.

checking [42]. Besides the CPA algorithm, Figure 7 depicts two additional algorithms as examples of further implemented verification approaches: **counterexample-guided abstraction refinement (CEGAR)** [59] and k-induction [27]. Both of these algorithms delegate parts of their work to the CPA algorithm. For example, CPAchecker can be configured to perform predicate abstraction [77] with counterexample-guided abstraction refinement by combining the CEGAR algorithm with a CPA that implements predicate abstraction. CPAchecker can also be configured to perform k-induction with the auxiliary-invariant generation, by combining the k-induction algorithm with a program analysis that produces invariants. For a detailed and formal discussion on how these approaches are implemented in CPAchecker, we refer the reader to the literature [28] and briefly introduce the concept of k-induction, only because it is a non-trivial but integral component of one of the validation approaches that we will later present.

k-Induction. To obtain unbounded proofs of safety, k-induction combines techniques from bounded model checking [50] with induction. Consider a verification task that contains an unbounded loop and a candidate invariant P for that task. It is possible (a) to check using a bounded model check with bound $k = 1$ whether a program path of length $k = 1$ exists for which P is violated, but this check cannot prove the absence of longer counterexample paths. However, it is possible (b) to prove that P is an invariant using induction if P is inductive, i.e., if P holds before a given loop iteration, P also holds after that iteration, by taking (a) as the base case of the induction proof and (b) as the inductive-step case.

An extension to greater values of k lifts (1-)induction to k-induction, where the invariant P is asserted not only before one loop iteration, but before each of k consecutive loop iterations in the step case to conclude that it also holds after the kth loop iteration. For $k > 1$, $(k-1)$-inductiveness implies k-inductiveness. In practice, k-induction may therefore succeed more often to prove correctness than $(k-1)$-induction [127], because k-induction uses a stronger induction hypothesis. A drawback of k-induction is that the approach cannot succeed if P is not k-inductive for any k. It is therefore desirable to strengthen P with auxiliary invariants to try making the assertion inductive.

KI⟲DF and KI⟲KI [27] are two k-induction techniques that are implemented in CPAchecker and use auxiliary invariants. In both techniques, an invariant generator runs in parallel to the k-induction procedure and successively provides invariants that are then used to strengthen the induction hypothesis. As time progresses, stronger invariants are generated, until the auxiliary invariants sufficiently strengthen the induction hypothesis to successfully prove the invariant P by induction. In KI⟲DF, the auxiliary-invariant generator is based on a data-flow analysis. Over time, the precision used by the analysis is increased, causing stronger invariants to be generated. In the k-induction technique KI⟲KI, the auxiliary-invariant generator is itself based on k-induction and attempts to prove invariants from a set of candidate invariants (either derived from a template or provided by a user). As time progresses, more confirmed candidates may become available as auxiliary invariants, until the induction hypothesis is strong enough to prove the safety property.

Example 5 (Verification with CPAchecker). Consider a verification task consisting of the program shown in Figure 1 and the specification from Figure 2. No feasible path to the call of the function `__VERIFIER_error(void)` in line 17 exists, because after the loop, the sum `s` of non-negative summands is always at least as great as its last summand `v`. Since the type of `s` is `unsigned int`, no overflow is caused by computing and storing the sum of at most `255` (maximum value of `n`) values, each of which is at most `255` (maximum value of `v`) itself, which in total is at most `65025`. Consequently, no feasible path to the call of the function `__VERIFIER_error(void)` in line 21 exists either. Therefore, the program satisfies the specification. CPAchecker is able to prove this by applying the KI⟲KI technique for k-induction and using a template for linear inequalities to produce candidate invariants for KI⟲KI: Knowing that $0 \leq i \leq 255$ due to the types of `i` and `n`,

CPAchecker can prove that the linear inequality $s \leq i \cdot 255$ is an invariant, and consequently also that $s \leq 65{,}025$ (and therefore also that s cannot overflow), which proves the program safe.

4.1.2 UAutomizer. The automata-based verification approach of UAutomizer [84] constructs a correctness proof as a sequence of automata. UAutomizer uses the concept of *Floyd–Hoare automata* (cf. Section 3.1.5) instead of an ARG. UAutomizer won the category *Overall* of the competition on software verification (SV-COMP) two times from 2012–2020.[6]

Architecture. Like CPAchecker, UAutomizer transforms the given program into a CFA and the given specification into an observer automaton, and uses the specification to determine whether a program path is an error path. The automaton product of the CFA and the observer automaton for the specification yields a new CFA that describes a formal language over the alphabet G, where G is the set of control-flow edges, and where the accepting states are defined by the observer automaton for the specification. Hence, the words accepted by this automaton are exactly those paths through the program that violate the specification, i.e., the error paths (cf. Section 2.1).

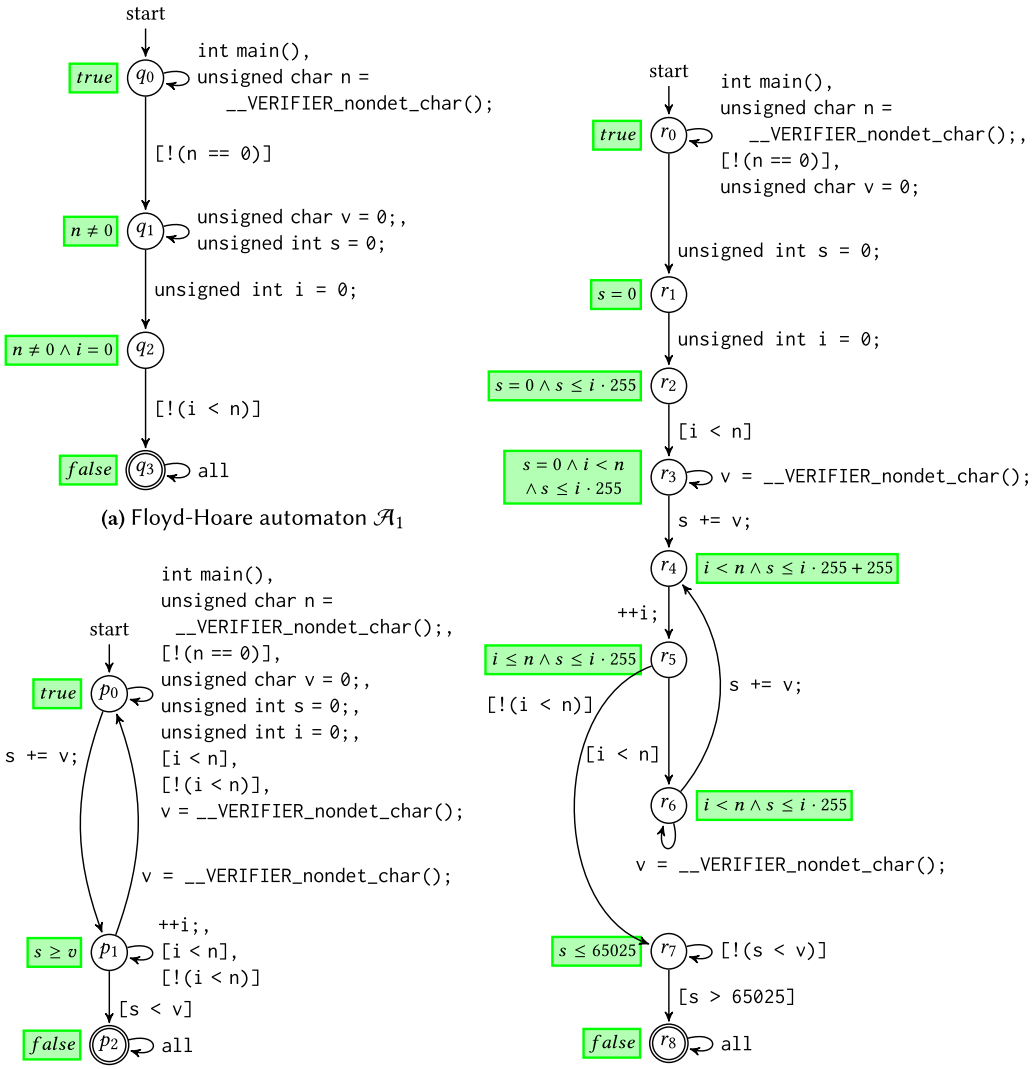
Verification Using Floyd–Hoare Automata. To solve a verification task, UAutomizer iteratively constructs Floyd–Hoare automata instead of an ARG. The Floyd–Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are constructed such that each automaton accepts only words that correspond to infeasible paths. If, at some point of this iterative process, the union of the languages of these automata becomes a superset of the language accepted by the product of the CFA and the observer automaton for the specification, the verification is complete and the constructed Floyd–Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ represent a correctness proof for the program.

Given a CFA $\mathcal{A}_P$ and the Floyd–Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ from the above-mentioned correctness proof, the following approach can be used to construct invariants: First, an automata-theoretical product of the automata $\mathcal{A}_P$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is constructed. We make sure that in the product construction no Floyd–Hoare automaton is blocking and make each automaton total beforehand (that is, they are observer automata). The totalization is implemented by implicitly adding for each missing outgoing transition (in the automata in Figure 8) an outgoing transition whose target is the initial state. Because the initial state is labeled by *true*, the totalized automata are still Floyd–Hoare automata. The states of the product are tuples of the form $(l, s_1, \ldots, s_n)$ where the first component is a program location of the CFA, and the $(i+1)$-th component s_i is a state of the Floyd–Hoare automaton $\mathcal{A}_i$. Each tuple in the product is annotated by a formula that is the n-ary conjunction of the invariants of all s_i, that is, the annotation of the tuple $(l, s_1, \ldots, s_n)$ is the conjunction $\bigwedge_{i=1}^{n} \varphi_{s_i}$. Then, the invariant for a location l is computed as the disjunction of all annotations of those tuples that are reachable in the product and where the first component is location l.

Example 6 (Verification with UAutomizer). The three Floyd–Hoare automata depicted in Figure 8 are a proof that the program whose CFA is depicted in Figure 1 satisfies the specification depicted in Figure 2. The observer automaton (Figure 2) for the specification considers the locations after the function `__VERIFIER_error(void)` was called, i.e., l_{18} and l_{22}, accepting. The three Floyd–Hoare automata are a proof of correctness, because each word that is accepted by the CFA is also accepted by $\mathcal{A}_1$, $\mathcal{A}_2$, or $\mathcal{A}_2$.

Intuitively, the Floyd–Hoare automaton $\mathcal{A}_1$ says that we cannot leave the while loop without passing the body at least once. The Floyd–Hoare automaton $\mathcal{A}_2$ says that after running the while loop (at least) once, the value of `s` is not smaller than the value of `v` and hence the program cannot reach the first call of the `__VERIFIER_error(void)` function. The Floyd–Hoare automaton $\mathcal{A}_3$ says that $s \leq i \cdot 255$ is a loop invariant and since the loop counter `i` is bounded by an unsigned

[6] https://ultimate.informatik.uni-freiburg.de/Automizer.



(a) Floyd-Hoare automaton $\mathcal{A}_1$

(b) Floyd-Hoare automaton $\mathcal{A}_2$

(c) Floyd-Hoare automaton $\mathcal{A}_3$

Fig. 8. Proof that the program whose CFA is depicted in Figure 1 satisfies the specification depicted in Figure 2. We construct a product of these automata in order to obtain invariants for the program.

char, the value of `s` is bounded by 65,025 and the program cannot reach the second call of the `__VERIFIER_error(void)` function.

Table 2 shows the annotations for the reachable states in the product of the automata from Figure 8, and Table 3 shows the invariants that we obtain for the program depicted in Figure 1. We do not show the locations l_6, l_{24}, and l_{25}, because UAutomizer detects beforehand that these locations do not occur on any path from the initial location to an error location and assigns the invariant *true* to these locations. In Table 2 we leave out all successors of all tuples that are annotated with *false* (l_{18} and l_{22}). Because each Floyd–Hoare automaton has a self-loop for locations with the invariant *false*, each successor of a tuple annotated by *false* is also annotated by *false*.

Table 2. Reachable States in the Product Automaton of $\mathcal{A}_P$, $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ Together with Their Annotation

State	Annotation
(l_3, q_0, p_0, r_0)	$true$
(l_4, q_0, p_0, r_0)	$true$
(l_5, q_0, p_0, r_0)	$true$
(l_8, q_1, p_0, r_0)	$n \neq 0$
(l_9, q_1, p_0, r_0)	$n \neq 0$
(l_{10}, q_1, p_0, r_1)	$n \neq 0 \wedge s = 0$
(l_{11}, q_2, p_0, r_2)	$n \neq 0 \wedge i = 0 \wedge s = 0 \wedge s \leq i \cdot 255$
(l_{16}, q_3, p_0, r_0)	$false$
(l_{12}, q_0, p_0, r_3)	$s = 0 \wedge i < n \wedge s \leq i \cdot 255$
(l_{13}, q_0, p_0, r_3)	$s = 0 \wedge i < n \wedge s \leq i \cdot 255$
(l_{14}, q_0, p_1, r_4)	$s \geq v \wedge i < n \wedge s \leq i \cdot 255 + 255$
(l_{11}, q_0, p_1, r_5)	$s \geq v \wedge i \leq n \wedge s \leq i \cdot 255$
(l_{12}, q_0, p_1, r_6)	$s \geq v \wedge i < n \wedge s \leq i \cdot 255$
(l_{13}, q_0, p_0, r_6)	$i < n \wedge s \leq i \cdot 255$
(l_{14}, q_0, p_1, r_4)	$s \geq v \wedge i < n \wedge s \leq i \cdot 255 + 255$
(l_{16}, q_0, p_1, r_7)	$s \geq v \wedge s \leq 65{,}025$
(l_{17}, q_0, p_2, r_0)	$false$
(l_{20}, q_0, p_0, r_7)	$s \leq 65{,}025$
(l_{21}, q_0, p_0, r_8)	$false$

Table 3. Invariants for the Program Depicted in Figure 1

Location	Invariant
l_3	$true$
l_4	$true$
l_5	$true$
l_8	$n \neq 0$
l_9	$n \neq 0$
l_{10}	$n \neq 0 \wedge s = 0$
l_{11}	$(n \neq 0 \wedge i = 0 \wedge s = 0 \vee s \geq v \wedge i \leq n) \wedge s \leq i \cdot 255$
l_{12}	$(s = 0 \vee s \geq v) \wedge i < n \wedge s \leq i \cdot 255$
l_{13}	$i < n \wedge s \leq i \cdot 255$
l_{14}	$s \geq v \wedge i < n \wedge s \leq i \cdot 255 + 255$
l_{16}	$s \geq v \wedge s \leq 65{,}025$
l_{17}	$false$
l_{20}	$s \leq 65{,}025$
l_{21}	$false$

4.2 Result Validation Based on Violation Witnesses

Violation witnesses are verification witnesses that represent error paths, i.e., paths through the program source code that violate the specification (Section 3.2.2).

4.2.1 Principles. A violation-witness-based result validator can attempt to validate the verification results as FALSE if the result is supported by a violation witness. In 2019, four implementations of such validators existed: CPAchecker, UAutomizer, CPA-witness2test, and FShell-witness2test.

Conceptually, all of these validators are based on the principle of witness-based result validation as described in Section 3.5.2. We classify the four validators into two categories, namely the category of *static* (*model-checking-based*) validators, and the category of *dynamic* (*execution-based*) validators, as indicated in Figure 4.

Static Validation. CPAchecker and UAutomizer are *static* validators, because they are based on purely static analysis and do not actually execute the program to confirm a violation. The advantages of static validators are that (1) they do not strictly require precise witnesses but can also be used with imprecise witnesses and can even be used to refine them, using witness refinement (cf. Section 3.5.3), because they can use model-checking techniques to detect invariants or even find concrete value assignments for program variables, (2) they can be used to validate results for verification tasks of systems with arbitrary target architectures, independent from the environment the validator is executed in, because they do not need to execute the analyzed program, and (3) they can be used for arbitrary specifications and are not limited to specifications with finite counterexamples (cf. dynamic validators). The main disadvantage of static validators is that accurately modeling all features of a complex programming language (such as C) is often difficult, and static validators therefore may exhibit the same imprecisions that also contribute to false alarms in verifiers. Thus, they may be less trustworthy than dynamic validators, which actually run the analyzed program to confirm a violation.

Example 7 (Violation-witness Construction, Validation, and Refinement). We illustrate how violation witnesses are constructed, validated, and refined across verifiers: We start with an overview of an example scenario and then describe the process of producing, consuming, and refining violation witnesses in more detail.

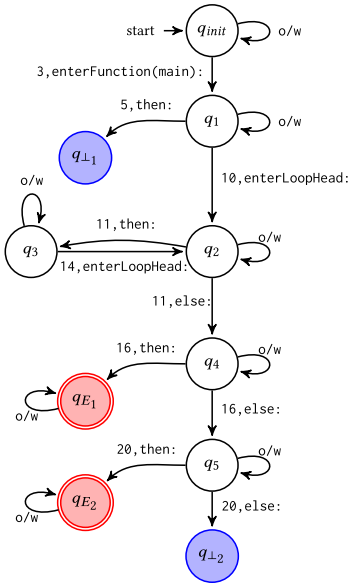
In this example, we first run three verifier instances in sequence. Each of them takes the verification task that consists of the program depicted in Figure 9(a) and the specification shown in Figure 2 as input, and produces a violation witness. The program in Figure 9(a) differs from the program in Figure 1(a) (and described in Section 2.1) in only one line: In line 9 of the original program, variable `s` is declared as type `unsigned int`, whereas, in line 9 of the modified program, `s` is declared as type `unsigned char`; therefore, the only difference between the CFA of the original program (cf. Figure 1(b)) and the CFA of the modified program is that the label of the CFA edge between l_9 and l_{10} is changed to `unsigned char s = 0`. The specification requires that no call to the function `__VERIFIER_error()` must be reachable from the program entry. The program violates this specification: Recall from Section 2.1 that the program attempts to compute the sum of a number of input values. However, the variable `s` that is used to store the computed sum is now declared to be of type `unsigned char`, which in our setting means that it is only 8 bits wide and can only store values between `0` and `255`. It is therefore not suitable for the task of storing the sum of up to `255` values in the range of `0` to `255` and is susceptible to arithmetic overflows. As a result, it is possible —depending on the actual input values— that in line 16, the condition `s < v` actually holds, and the function `__VERIFIER_error()` is called, thereby violating the specification.

In this example scenario, all three verifiers are configured as a composite CPA (cf. Section 2.2.2). The first verifier runs an analysis that considers only control-flow information and does not track any variable values, and produced Witness 1 (Figure 9(b)). The second verifier then takes this witness and the verification task as input, runs an analysis based on an interval domain [64], and produces the violation witnessed in Figure 9(c). In the third step, we run a test-case generator [19, 26, 35] that takes the violation witness from Figure 9(c) and the verification task as input and produces the test vector that is represented by the witness in Figure 9(d).

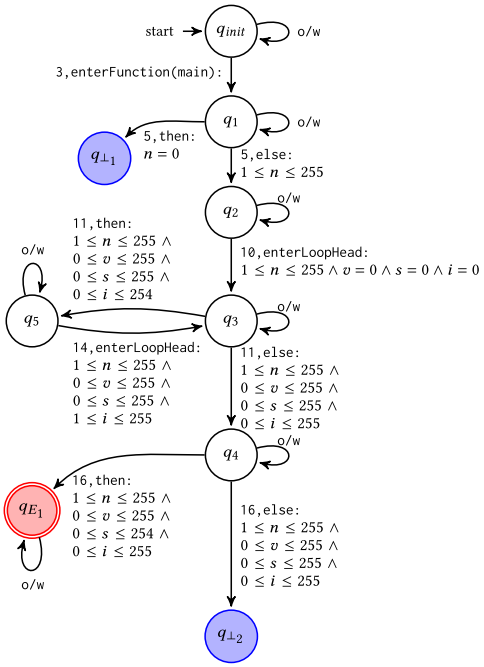
Verification. The Composite CPA used by the first verifier is composed of a Location CPA that tracks the program counter and an observer analysis (i.e., an Automaton CPA for an observer automaton,

```
1 extern void __VERIFIER_error(void);
2 extern unsigned char __VERIFIER_nondet_char(void);
3 int main() {
4   unsigned char n = __VERIFIER_nondet_char();
5   if (n == 0) {
6     return 0;
7   }
8   unsigned char v = 0;
9   unsigned char s = 0;
10  unsigned int  i = 0;
11  while (i < n) {
12    v = __VERIFIER_nondet_char();
13    s += v;
14    ++i;
15  }
16  if (s < v) {
17    __VERIFIER_error();
18    return 1;
19  }
20  if (s > 65025) {
21    __VERIFIER_error();
22    return 1;
23  }
24  return 0;
25 }
```

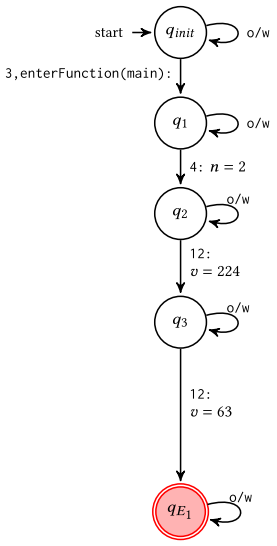
(a) Unsafe program linear-inequality-inv-b.c (adapted from Fig. 1a)



(b) Witness 1



(c) Witness 2



(d) Witness 3

Fig. 9. Example C program with a bug (a) and violation witnesses for it (b, c, and d).

cf. Figure 7) that tracks the control state of the observer automaton for the specification. Abstract states of this composite analysis are tuples $(l, (s, \psi_s))$, where l represents the current location in the CFA, i.e., the component abstract state tracked by the Location CPA, and (s, ψ_s) is the component abstract state tracked by the observer analysis, which consists of the current control state s of

the observer automaton for the specification and the current state-space restricting condition ψ_s. Because the specification is represented as an observer automaton, the observer analysis does not restrict the state space. This analysis will detect a violation if its abstract state is $(\cdot, (s_E, \cdot))$, i.e., if the observer analysis that monitors the observer automaton for the specification transitions into an accepting state. The initial state is $(l_3, (s_0, \mathit{true}))$, i.e., the CFA is in its initial location l_3 and the observer automaton is in its initial state s_0. The first witness automaton (Figure 9(b)) is produced by this verifier. The analysis marks the program entry in line 3 by writing an automaton transition with the source-code guard $(l_3, \texttt{int main()}, l_4)$ and the state-space guard *true*, which is described by the label `3,enterFunction(main):` in our graphical representation.[7] The analysis detects that taking the `then`-branch in line 5 cannot lead to a violation of the specification, and thus, writes a transition to the sink state $q_{\bot_1}$, and similarly for the `else`-branch in line 20. Because the analysis does not include any information about program variables, it is unable to eliminate any infeasible program path to the function-call operations in lines 17 and 21, and thus, it writes transitions to the accepting states q_{E_1} and q_{E_2} to represent each of these violations. Consequently, the resulting witness only overapproximates the set of feasible error paths: In fact, all of the error paths that lead to the violation in line 21 are actually infeasible, and there is no restriction on the values of program variables or the number of loop unrollings because the analysis is very imprecise.

Witness Refinement. In the next step of our example scenario, we use a verifier that runs an analysis based on an interval domain, and we give it as input the verification task and the witness from Figure 9(b), which was produced in the previous step. The Composite CPA we use for this analysis consists of the following component CPAs: A Location CPA to track the program counter, an observer analysis to track the control state of the observer automaton for the specification, a witness analysis (Automaton CPA for a witness automaton) to track the control state of the witness automaton, and an Interval CPA to track the values of variables using an interval domain. Abstract states of this Composite CPA are tuples $(l, (s, \psi_s), (q, \psi_w), e)$, where l and (s, ψ_s) again represent the component abstract states of the Location CPA and the observer analysis for the observer automaton for the specification, respectively, (q, ψ_w) is the component abstract state of the witness analysis that consists of the current control state q and state-space condition ψ_w from the witness automaton, and e is the component abstract state of the Interval CPA, that is, a mapping from the set X of program variables to intervals. Because we are consuming a violation-witness automaton, we will only consider specification violations where both the observer automaton for the specification and the witness automaton for the violation witness are in an accepting state. The initial state is $(l_3, (s_0, \mathit{true}), (q_{\mathit{init}}, \mathit{true}), \{n \to [-\infty, \infty], v \to [-\infty, \infty], s \to [-\infty, \infty], i \to [-\infty, \infty]\})$, i.e., the CFA is in its initial location l_3, the observer automaton for the specification is in its initial state s_0, the witness automaton is in its initial state q_{init}, and there is currently no information on variable values. From the initial program location l_3, the Location CPA only allows a transition via the CFA edge to l_4, which means that the first analyzed operation is the program entry in line 3. In the specification automaton, only the self transition on state s_0 labeled "*o/w*" matches; recall from Section 3.1 that such self-transitions only match if no other transitions are applicable, and that they impose no state-space restrictions. In the witness automaton, the transition from q_{init} to q_1 labeled `3,enterFunction(main):` matches. We do not gain any information on variable values. Therefore, the successor composite abstract state is $(l_4, (s_0, \mathit{true}), (q_1, \mathit{true}), \{n \to [-\infty, \infty], v \to [-\infty, \infty], s \to [-\infty, \infty], i \to [-\infty, \infty]\})$. Next, the analysis progresses via the operation on line 4, which declares the variable `n` of type `unsigned char` and initializes it via an input value by calling the function `__VERIFIER_nondet_char(void)`.

[7] While the token `enterFunction(main)` would already be sufficient to unambiguously describe the source-code guard, we always add the line number for the reader's convenience.

Hence, the new location in the CFA is l_5, the observer automaton for the specification again takes the self-transition and stays in s_0, the witness automaton also has no matching transition other than the self-transition *o/w* at q_1 and therefore stays in q_1, and the new interval abstract state is $\{n \to [0, 255], v \to [-\infty, \infty], s \to [-\infty, \infty], i \to [-\infty, \infty]\}$. At l_5, the CFA branches. We first consider the branch to l_6. The observer automaton for the specification stays in s_0 again, but the witness automaton has a matching transition to $q_{\bot_1}$. The interval analysis detects that due to the branching condition, $n \to [0, 0]$ and updates its successor component abstract state accordingly. However, since $q_{\bot_1}$ is a sinking state, the witness analysis that tracks the control state of the witness automaton will not produce any further successors on this branch, so we can eliminate all corresponding program paths and need not consider them any more. We now consider the branch from l_5 to l_8. Here, the observer automaton for the specification also stays in s_0, the witness automaton stays in q_1, and the new interval abstract state is $\{n \to [1, 255], v \to [-\infty, \infty], s \to [-\infty, \infty], i \to [-\infty, \infty]\}$. After the next two operations on lines 8 and 9, which declare the variables `v` and `s` of type `unsigned char` and initializes them both to `0`, the composite abstract state is $(l_{10}, (s_0, \mathit{true}), (q_1, \mathit{true}), \{n \to [1, 255], v \to [0, 0], s \to [0, 0], i \to [-\infty, \infty]\}$. The next operation, in line 10, declares the variable `i` of type `unsigned char` and initializes it to `0`. The CFA is then in location l_{11}, which is a loop head. Therefore, the witness-automaton transition from q_1 to q_2 matches, and the composite abstract state is $(l_{11}, (s_0, \mathit{true}), (q_2, \mathit{true}), \{n \to [1, 255], v \to [0, 0], s \to [0, 0], i \to [0, 0]\}$. Next, we follow the branch from the loop head l_{11} to l_{12}, i.e., into the loop, which leads over the locations l_{12}, l_{13}, and l_{14} and the witness-automaton state q_3 eventually back to the loop head l_{11} and the witness-automaton state q_2. We assume that the analysis is able to compute a fixed point, but at the loss of precision: When entering the loop, `i` must be lower than `n`, which is at most `255`, so `i` must be at most `254`, and since `i` is incremented within the loop, it must be between `1` and `255` at the end of each loop iteration. The most precise single abstract state in our current composite abstract domain that covers all reached states at the loop head l_{11}, however, is $(l_{11}, (s_0, \mathit{true}), (q_2, \mathit{true}), \{n \to [1, 255], v \to [0, 255], s \to [0, 255], i \to [0, 255]\}$. After reaching this fixed point for the loop, we continue with the branch from l_{11} to l_{16}, which matches the source-code guard `11,else` on the witness-automaton transition from q_2 to q_4. At this point, we encounter another branching. We first take the branch from l_{16} to l_{17}, which matches the source-code guard `16,then` on the witness-automaton transition from q_4 to q_{E_1}, but since the specification automaton still stays in s_0, which is not an accepting state, we continue to the following operation, which is the function call to `__VERIFIER_error(void)` on line 17. This operation matches the source-code guard `enterFunction(__VERIFIER_error)` on the observer-automaton transition from s_0 to the accepting state s_E. Because the witness automaton stays in the accepting state q_{E_1}, the analysis has now found a program path to a specification violation that is described by the input witness, and it writes a transition to the corresponding accepting control state q_{E_1} into its output witness (Figure 9(c)). We now follow the branch from l_{16} to l_{20}. For this operation, we compute the successor state $(l_{20}, (s_0, \mathit{true}), (q_5, \mathit{true}), \{n \to [1, 255], v \to [0, 255], s \to [0, 255], i \to [0, 255]\})$. We then encounter another branching. When attempting to compute the successor abstract state via the branch from l_{20} to l_{21}, the interval component will detect that all program paths along this branch are infeasible. Therefore, no successor abstract state for this branch is computed. Along the other branch from l_{20} to l_{24}, the source-code guard `20,else` on the witness-automaton transition to $q_{\bot_2}$ matches, so that the analysis does not continue along this branch either, and the state-space exploration is complete. Because the analysis did not encounter any violation states via the `else`-branch in line 16, it writes a corresponding transition to a sinking state into its output witness. The new witness (Figure 9(c)) is more precise than the input witness (Figure 9(b)), as it does not contain the infeasible error paths to line 21 and puts restrictions on variable values, but it is still an

overapproximation of the set of feasible error paths. For example, it contains infeasible error paths that never enter the loop and, therefore, cannot trigger the overflow that makes the violation in line 17 reachable.

Execution-based Witness Validation. In the third step of our example scenario, the violation witness from Figure 9(c) is used to restrict a test-case generator [26] to derive a specific test vector for the program path to the call to `__VERIFIER_error(void)` in line 17. The test vector is derived by extracting a satisfying assignment of the formula that represents the program path to l_{18}. The third witness automaton (Figure 9(d)) represents the result of the test-case generation, i.e., a test vector, which contains all the input data necessary to execute a test of the program that triggers the described violation of the specification. This witness precisely represents exactly one feasible error path, and is therefore an underapproximation of the set of feasible error paths, because there are also other paths that would lead to a violation, for example with more than two loop iterations or with a different pair of summands.

Dynamic Validation. We call CPA-witness2test and FShell-witness2test *dynamic* validators, because they perform only very light-weight static analysis to extract a test vector from a violation witness, and then compile, link, and execute the program with a corresponding test harness to dynamically observe whether the specification is actually violated during the execution. The advantages of dynamic validators are that (1) they can be much more efficient than static validators, because they do not require expensive model-checking techniques, (2) they can be more precise than static validators, because a violation that can be observed during an actual execution of a program undeniably exists, (3) an executable produced by a dynamic validator can be used by developers to analyze a bug by applying standard tools they already know and are well-trained in, such as debuggers, and (4) a test harness produced by a dynamic validator can directly be used by developers to improve their test suite and prevent regressions once the bug is fixed. There are, however, also some disadvantages: A dynamic validator requires as input a witness that represents a test vector, i.e., a witness that specifies concrete value assignments for all program inputs, which may not be available from all verifiers. To obtain a suitable witness, it is possible to first apply witness refinement to the original witness, but, because witness refinement uses the expensive techniques of static validators, this solution negates the first advantage of dynamic validators over static validators (i.e., their efficiency). The second disadvantage of dynamic validators is that they require a concrete and secure execution environment[8] that matches the target environment of the analyzed system, whereas static validators can, conceptually, also be used in any (unrelated) other execution environment. Lastly, dynamic validators can only confirm a violation if the time required to execute the program and trigger the bug is finite (and reasonably brief). If, for example, the specification is that the program must always terminate, a validator for a violation would need to confirm that there is a path that does not terminate; this cannot be observed from a finite execution.

Example 8 (Execution-based Validation). We now demonstrate execution-based validation of verification results as it would be performed by the validator CPA-witness2test when applied to the verification task composed of the program from Figure 9(a) and the specification from Figure 2, and the third and most precise witness from our previous example shown in Figure 9(d). To extract the input values from the witness, match them to the input functions of the program, and generate the test harness depicted in Figure 10, the validator first runs a light-weight program analysis configured as a Composite CPA composed of the following components: a Location CPA that tracks the program counter, an observer analysis that tracks the control state of the observer automaton

[8] Test-suite validators (such as TestCov [41]) can be used for the safe and secure execution of tests.

```
1 struct _IO_FILE;
2 typedef struct _IO_FILE FILE;
3 extern struct _IO_FILE *stderr;
4 extern int fprintf(FILE *__restrict __stream, const char *__restrict __format,
  ↪ ...);
5 extern void exit(int __status) __attribute__ ((__noreturn__));
6 void __VERIFIER_error() {
7   fprintf(stderr, "cpa_witness2test: violation\n");
8   exit(107);
9 }
10 unsigned char __VERIFIER_nondet_char() {
11   static unsigned int test_vector_index = 0;
12   unsigned char retval;
13   switch (test_vector_index) {
14     case 0: retval = (unsigned char)2; break;
15     case 1: retval = (unsigned char)224; break;
16     case 2: retval = (unsigned char)63; break;
17   }
18   ++test_vector_index;
19   return retval;
20 }
```

Fig. 10. Test harness generated from the witness of Figure 9(d) for the C program of Figure 9(a).

for the specification, and a witness analysis that tracks the control state of the witness automaton for the violation witness. The analysis traces the program paths that are described by the witness automaton and matches the values of input variables on these paths to the corresponding input functions of the program in the correct order. In the example, the automaton specifies that in line 4, the value assigned to the variable `n` (which controls how many further input values will be read) should be `2`; that in the first loop iteration in line 12, the value assigned to the variable `v` should be `224`; and that in the second loop iteration in line 12, the value assigned to the variable `v` should be `63`. Consequently, the validator produces the test harness listed in Figure 10, which contains an implementation of the input function `__VERIFIER_nondet_char(void)` that returns exactly those values in that order, and an implementation of the function `__VERIFIER_error(void)` that, if called, allows the validator to detect the specification violation through a custom program output. In the next step, the validator compiles and links the source code of the C program and the produced test harness, and executes the resulting program. As expected, the validator can observe the specification violation during execution, because the sum of `224` and `63` is `287`, which exceeds the value range of the type `unsigned char` of variable `s` and therefore wraps around to the value `31`. Because `31` is less than the last input value `63`, the function `__VERIFIER_error()` is called at line 17. The validator detects this function call and confirms the verification result.

4.2.2 Tool Implementations. We implemented violation-witness-based result validation in the two static validators CPAchecker and UAutomizer, and in the two dynamic validators CPA-witness2test and FShell-witness2test.

Static Violation-Witness-Based Result Validation with CPAchecker. Figure 11 shows a section of the architecture of CPAchecker that implements violation-witness-based result validation. The left side of the figure shows the inputs, consisting of the verification task (i.e., program and specification) and the violation witness. The program is parsed and converted into a CFA, the specification into an observer automaton, and the violation witness into a witness automaton. Then, the CPA algorithm is run with a composite program analysis that is composed of at least a Location CPA and two

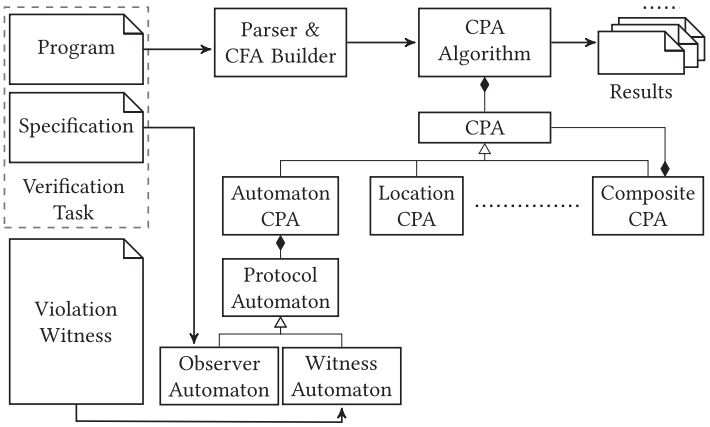


Fig. 11. The architecture of the violation-witness-based result validator implemented in CPAchecker.

Automaton CPAs, one for the observer automaton for the specification and one for the witness automaton for the violation witness. As mentioned in Section 4.1.1, further CPAs are available and can optionally be added to the composition to enhance the capabilities of the program analysis; in our evaluation (cf. Section 6), we add predicate analysis [28] and explicit-state model checking [42].

After the CPA algorithm completes, the computed results are delivered. This may either be a simple confirmation (or refutation) of the validated result, or it may be a refined violation witness if the validated result was confirmed and the validation process was able to add information to an imprecise input witness. The witness is confirmed if the observer automaton and the witness automaton *both* reached one of their accepting control states.

Static Violation-Witness-Based Result Validation with UAutomizer. While we formalized witness-based result validation using the concept of configurable program analysis, other approaches are also applicable: UAutomizer performs the validation in two steps. In the first step, a new CFA is constructed that represents those paths of the original CFA that comply with the source-code guards of the witness automaton, i.e., the new CFA is constructed as a product of the original CFA and the witness automaton. The states of this product automaton are pairs (l, q), where l is a location of the original CFA and q is a control state of the witness automaton. The product contains a transition from (l, q) to (l', q') labeled with op if

- (l, op, l') is a CFA edge and
- $q \xrightarrow{(S,\psi)} q'$ is a transition in the witness automaton such that $(l, op, l') \in S$.

In the second step, UAutomizer verifies if the resulting CFA satisfies the specification using its automata-theoretic verification approach [84]. The witness is confirmed if a violation of the specification is found, that is, the observer automaton for the specification reached an accepting control state and the new CFA also reached an accepting control state.

Dynamic Violation-Witness-Based Result Validation with CPA-witness2test. Figure 12 shows the witness2test-workflow for violation-witness-based result validation with dynamic validators: The validator receives as input the verification task and the violation witness produced by a verifier, and synthesizes from these inputs a test harness for the program. This test harness and the program source code are compiled and linked with a C compiler to produce an executable program, which is then executed. Assuming that the witness represents a precise test vector and the validator correctly translates the witness into a test harness, if the validator observes a violation, then

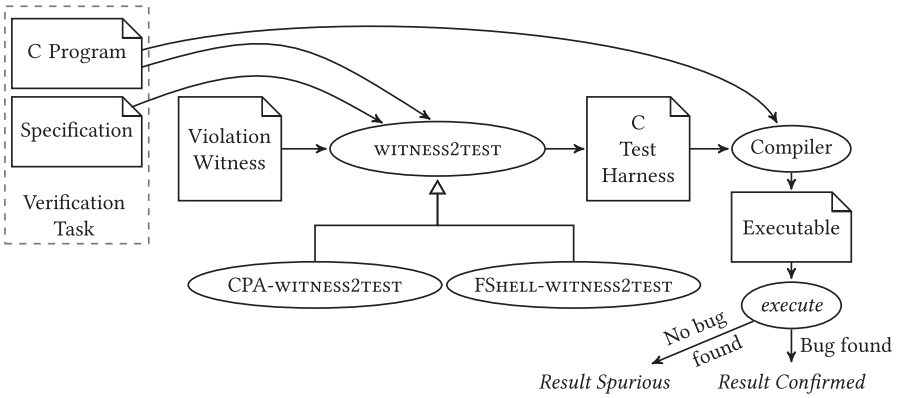


Fig. 12. Flow of violation-witness-based result validation with dynamic validators (witness2test).

the bug found by the verifier and described by the witness is realizable and the result is therefore confirmed; otherwise, it is refuted. Because an executable program that triggers an actual bug is always available after a successful validation, the developer can immediately start debugging, for example by running the executable with a debugger like GDB.

For CPA-witness2test, the step of extracting the test vector from the verification task and witness is conceptually similar to the validation performed by static validators, except that the static analysis is used only to assign the input values from the witness to the correct input functions of the program, not to perform any further semantic reasoning about the program. In fact, CPA-witness2test uses also the architecture displayed in Figure 11 for the step of matching the input values from the witness to the input functions of the program. Unlike the static validators, however, it does not add further CPAs and its result in this step is the test harness.

Dynamic Violation-Witness-Based Result Validation with FShell-witness2test. The key design principle of FShell-witness2test, on the other hand, is independence from existing verification infrastructure: the results of FShell-witness2test are by design unbiased towards any existing software-analysis framework. Consequently, FShell-witness2test is another example that shows that while we formalize witness-based result validation using the CPA concept, implementations that follow other paradigms are also applicable in practice. The architecture of FShell-witness2test consists of two major parts: (1) a Python-based processor of the violation witness and the program source code, using `pycparser`,[9] to generate a test vector in a format compatible with FShell [89] (hence the name of the validator), and (2) a Perl script to convert such a test vector into a test harness that can be compiled and linked with the input program. For a given violation witness and verification task, FShell-witness2test first parses the specification to determine the expected type of violation. The witness and the C program are then handed to the Python-based processor. Because `pycparser` cannot handle various GCC extensions, input programs are preprocessed and sanitized by performing text replacement and removal. FShell-witness2test then obtains the abstract syntax tree and iterates over its nodes to gather data types and source locations of input-value assignments. Finally, FShell-witness2test builds a linear sequence of states from the witness automaton. Traversing this sequence, any match of line numbers against the input-value assignments triggers an attempt to extract values from assumptions in the witness. If the assumption represents a precise value assignment, an input value is recorded.

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

4.3 Result Validation Based on Correctness Witnesses

Correctness witnesses are verification witnesses that represent the artifacts of a proof that a program satisfies a specification, i.e., invariants for certain program locations that are intended to help reconstruct a correctness proof (Section 3.2.3).

4.3.1 Principles. The program analysis of a correctness-witness-based result validator checks if the given invariants indeed hold at their corresponding abstract program states; validation of a correctness witness fails if the validator refutes the invariant φ for an abstract program state or if it detects a violation of the specification, i.e., a feasible error path.

There are only two differences between violation-witness validation and correctness-witness validation:

- Violation-witness-based result validation uses assumptions at the witness automaton's transitions to constrain the state space; a correctness witness does not constrain the state space but contains at each control state in the witness automaton a state invariant.
- Violation-witness-based result validation attempts to replay an error path through the program, while correctness-witness-based result validation tries to replay the correctness proof: after confirming a witness invariant, it may use it as an auxiliary lemma to prove the correctness of the program or further witness invariants.

4.3.2 Tool Implementations. Currently, two implementations of correctness-witness-based result validators exist. We describe the two different strategies employed by CPAchecker and UAutomizer (out of the many possible strategies to implement a validator), which are implemented in two different verification frameworks to demonstrate the potential and flexibility of the approach.

CPAchecker's Correctness-Witness-Based Result Validator. Like the CPAchecker-based verifier from Section 4.1.1, the CPAchecker-based validator for correctness witnesses uses the KI↺KI technique for k-induction. In a preparatory step, the invariants are extracted from the correctness witness and mapped to their corresponding program locations. By design, a witness may be imprecise, therefore it is possible that an invariant is mapped to several program locations. The invariant generator then uses these invariants as candidate invariants, and, if it is able to prove the inductiveness of such a candidate invariant, it supplies it as an invariant to the main k-induction procedure. If, on the other hand, the invariant generator is able to refute a candidate at all program locations described by its corresponding state in the witness automaton, the validation fails, i.e., refutes the result.

One of the advantages of using k-induction for correctness-witness-based result validation is that for non-trivial software-verification tasks, k-induction is known to perform well only if it is supplied with the necessary auxiliary invariants [27, 99]. By design, all techniques that are implemented in CPAchecker to generate its own auxiliary invariants (e.g., from data-flow analysis) are turned off for the validation. Consequently, the validator's success in confirming a proof result depends on the quality of the invariants given by the witness.

Within each iteration of the k-induction procedure of the KI↺KI technique's invariant generator, CPAchecker will try to refute each invariant provided by the witness by finding a counterexample of the current length k before trying to prove its correctness. Hence, CPAchecker is guaranteed to find incorrect invariants with counterexamples that are at most as long as the value of k is required to prove that the program conforms to its specification, and it is also guaranteed to only use supplied invariants that it can prove to be correct. CPAchecker does not guarantee, however, that it will detect incorrect witness-supplied invariants if the length of their shortest counterexample exceeds the value of k required to prove that the program itself is correct. This is a design decision of the implementation, not a limitation of the concept of correctness witnesses or the CPAchecker

framework. To instead exhaustively confirm or refute all provided invariants, CPAchecker could be changed to simply defer checking the correctness of the program until the KI⇜⇝KI invariant generator has processed all invariants. The reasoning for the design decision without exhaustive checking was to not discourage developers of verifiers from producing invariants that k-induction might struggle with and cause exhaustive checks to time out. Moreover, if proving the correctness of a program requires an auxiliary invariant and the witness provides a correct one, the witness can already be considered useful, even if not all of its contents are checked exhaustively. For use cases where exhaustive proof or refutation of all invariants is desired, an alternative implementation is provided by UAutomizer.

Ultimate Automizer's Correctness-Witness-Based Result Validator. To validate a proof result, UAutomizer verifies the given program and considers each invariant provided by the correctness witness as an additional specification. Each of the specifications (the additional specifications from the invariants and the original specifications) is checked in the order of their occurrence in the program, and if correct, can be assumed while checking specifications occurring later in the program. UAutomizer confirms the result if the original specification and all specifications derived from witness invariants hold. If one specification cannot be confirmed, the validation fails (result refuted). In case we do not want to validate the witness as a whole but would like to point out incorrect invariants individually, we can check each specification individually without assuming validity for any other specification.

Converting a witness invariant into an additional specification is implemented as follows. First, an observer analysis matches the program CFA against the witness to obtain a partial map f from program locations to witness invariants. In a second step, the CFA is modified as follows. For each location l for which the mapping f is defined, UAutomizer

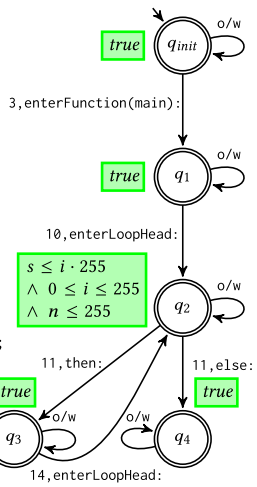
- adds a new location l',
- adds a new edge $(l, op_{f(l)}, l')$ where $op_{f(l)}$ is the assume operation that assumes the invariant $f(l)$ that was mapped to l,
- adds a new edge $(l, op_{\neg f(l)}, l_{\text{err}})$, where $op_{\neg f(l)}$ is the assume operation that assumes the negation of the invariant $f(l)$ and l_{err} is a location whose reachability is forbidden by the original specification, and
- replaces each outgoing edge of the form (l, op, l'') with an edge (l', op, l'').

The resulting CFA is verified as described in Section 4.1.2.

Example 9 (Correctness Witnesses). We illustrate the idea of correctness-witness validation using two short C programs listed in Figure 13(a) (taken from Figures 1(a)) and 13(c) (taken from Figure 9(a)), an example correctness-witness automaton shown in Figure 13(b), and the specification from Figure 2, which forbids reachable calls to the function `__VERIFIER_error(void)`. The first of the two C programs (Figure 13(a)) differs from the second C program (Figure 13(c)) only in one line: While variable `s` is declared with type `unsigned int` in line 9 of the first program, it is declared with type `unsigned char` in line 9 of the second program. As a result, the first program satisfies the specification, while the second program violates it, because `s` is susceptible to arithmetic overflows (cf. Example 7).

One way to prove that neither of the calls to the function `__VERIFIER_error()` in line 17 and line 21 is reachable in the first program would be to find an upper bound for the number of loop iterations and then unroll the loop, performing a bounded model check [50]. While this would still be feasible in our example due to the small types and simple structure that we chose for ease of presentation, it would already be expensive; for larger loop bounds, this strategy becomes infeasible, and for unknown loop bounds, it is impossible. Another—unbounded—way to prove

```
1 extern void
  ↪ __VERIFIER_error(void);
2 extern unsigned char
  ↪ __VERIFIER_nondet_char(void);
3 int main() {
4   unsigned char n =
    ↪ __VERIFIER_nondet_char();
5   if (n == 0) {
6     return 0;
7   }
8   unsigned char v = 0;
9   unsigned int  s = 0;
10  unsigned int  i = 0;
11  while (i < n) {
12    v = __VERIFIER_nondet_char();
13    s += v;
14    ++i;
15  }
16  if (s < v) {
17    __VERIFIER_error();
18    return 1;
19  }
20  if (s > 65025) {
21    __VERIFIER_error();
22    return 1;
23  }
24  return 0;
25 }
```




```
1 extern void
  ↪ __VERIFIER_error(void);
2 extern unsigned char
  ↪ __VERIFIER_nondet_char(void);
3 int main() {
4   unsigned char n =
    ↪ __VERIFIER_nondet_char();
5   if (n == 0) {
6     return 0;
7   }
8   unsigned char v = 0;
9   unsigned char s = 0;
10  unsigned int  i = 0;
11  while (i < n) {
12    v = __VERIFIER_nondet_char();
13    s += v;
14    ++i;
15  }
16  if (s < v) {
17    __VERIFIER_error();
18    return 1;
19  }
20  if (s > 65025) {
21    __VERIFIER_error();
22    return 1;
23  }
24  return 0;
25 }
```

(a) Safe program (from Fig. 1a) **(b)** Witness automaton **(c)** Unsafe program (from Fig. 9a)

Fig. 13. Example C programs (a and c) and a potential correctness witness (b), with the only difference between the two programs (line 9) highlighted.

that the program is safe is to prove that $s \le 65{,}025$ is an invariant of the loop from lines 11 to 15. This invariant is not inductive for the loop, however: It holds trivially before the loop, where $s = 0$, but the loop body does not guarantee that the invariant is preserved. Even strengthening this invariant to $s \le 65{,}025 \wedge i \le 255$ (which should be simple for any verifier that understands C types and knows that in our target architecture, type `unsigned char` is 8 bits wide) is not inductive. The predicate $s \le i \cdot 255 \wedge 0 \le i \le 255 \wedge n \le 255$, on the other hand, is an inductive invariant of the loop, but finding such an invariant is usually difficult, and depends on the verification strategy: it is, therefore, the critical step in solving this verification task.

A verifier that successfully proves the safety property for the program may then export a correctness witness. If the correctness witness contains the invariant $s \le i \cdot 255 \wedge 0 \le i \le 255 \wedge n \le 255$, a witness validator using the witness should be able to easily confirm the proof. Figure 13(b) displays a graphical representation of such a correctness witness. The automaton starts in an initial control state q_{init}. The witness assigns the invariant *true*. It is allowed to proceed to state q_1 if the control flow enters the `main` function of the program. As long as this transition is not possible, the automaton remains in state q_{init} via the self-transition "otherwise" (*o/w*). From q_1, the automaton is allowed to proceed to q_2 if the control-flow enters the loop head; otherwise, it remains in q_1 via the self-transition *o/w*. From q_2, the automaton can proceed to control state q_3 if the condition of the `while` loop in line 11 is true (the `then`-case), or to state q_4 if the condition in line 11 is false (the `else`-case). As long as none of these transitions are possible, the automaton remains in control state q_2 via the self-transition *o/w*. The automaton proceeds back from state q_3 to q_2 after the program operation in line 14; as long as this is not possible, the automaton will stay in q_3 via

its self-transition *o/w*. If the automaton is in control state q_4, it will stay there forever.[10] Control states q_{init}, q_1, q_3, and q_4 contain the trivial state invariant *true*. Control state q_2 specifies the invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$. Because state q_2 describes the loop head, a validator is able to prove (for example by induction) that the invariant holds at this program location, and can then use the invariant to prove the correctness of the program, thus validating the proof result.

If the invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$ is removed from the witness for state q_2, the witness is still valid (because *true* is an invariant). However, the k-induction-based validator will no longer confirm the proof because it lacks the information that is required to prove the correctness of the program, and it is not allowed to synthesize the required information itself. This is a design choice, in order to not confirm witnesses that are extremely weak (e.g., *true* everywhere).

Due to the structural similarity between the first program in Figure 13(a) and the second program in Figure 13(c), the witness in Figure 13(b) can also be matched with the second (unsafe) program. In this case, however, the loop invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$ does not imply that $s \geq v$, because the invariant is no longer sufficient to preclude overflows during the addition in line 13. In fact, if we conjoined $s \geq v$ to our invariant, it would still be a valid loop invariant in Figure 13(a) but not in Figure 13(c), where an overflow would be a counterexample to the inductiveness of $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255 \wedge s \geq v$. Hence, a validator will not be able to prove the correctness of the program using the loop invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$. Because each correctness-witness-based validation of a proof result also implicitly uses the safety property as an invariant, the validator can reject the witness by finding a feasible error path to line 17 as a counterexample to the specification, such as the one from Figure 9(d). The strength of the invariants determines the quality of the witnesses, but no particular strength is required. This example shows that correctness-witness-based validation can be more efficient than verification because it might be easier to (re-) verify that invariants indeed hold, while the verification needs to come up with the invariants. The task of finding useful invariants is in general considered one of the key challenges in software verification. Generalizing this approach allows for a lot of flexibility, because the more helpful the candidate invariants are, the less work has to be performed by the validator.

5 EXCHANGE-FORMAT SPECIFICATION

To store witness automata and exchange them across different verification tools and validators, we define an exchange format. Because automata are graphs, we use the existing graph format GraphML [51], which defines XML elements for edges (used in our format to model protocol-automaton transitions) and nodes (to model protocol-automaton control states).

The root element of a protocol-automaton GraphML document is the element `graphml`. The graph that models the protocol automaton is represented by the element `graph`, which is a child element of the root element `graphml`. We require that there is exactly one such `graph` element in the document. We model a control state of a protocol automaton using a `node` element. Each `node` element is a child element of the `graph` element and must specify a unique identifier (within the graph) for the control state using the attribute `id`. Analogously, we model a protocol-automaton transition using an `edge` element. Each `edge` element is a child element of the `graph` element and has the attributes `source` and `target`, both of which refer to `node` elements via their `id`s. Additional data can be attached to individual nodes and edges, and the graph itself, by adding `data` elements as child elements. The content of a `data` element is its value; each data element must specify its meaning via a `key` attribute. Each `key` that is used in the document must be defined using a

[10]The rest of the exploration does not matter for the witness, because the sole purpose of the witness is to attach the invariant at the right program location.

`key` element as a child element of the root element `graphml`. A `key` element must define the name of the key (using the attribute `attr.name`), the type of the values of `data` elements with this key (using the attribute `attr.type`), and whether data elements with this key are used on the `graph`, or on `node` or `edge` elements (using the attribute `for`), and a unique identifier for the key (using the attribute `id`. Valid values for the `attr.type` attribute in GraphML are `boolean`, `int`, `long`, `float`, `double`, and `string`. For protocol automata, we use the type `boolean` for boolean values, `int` for integer values, and `string` for other values. The `key` attribute of a `data` element refers to the value of the `id` attribute of the corresponding `key` element, not its name. A default value can be defined for each key as the content of a `default` element that is added to the enlargethispage8ptdesired `key` element as a child element.

Keys for `graph` Elements. The following keys are defined for `data` elements that are used to add information that concerns the witness as a whole, i.e., for `data` elements that are direct children of the `graph` element:

- `witness-type` is used to specify the witness type. A correctness witness is identified by the value `correctness_witness`, a violation witness is identified by the value `violation_witness`.
- `sourcecodelang` is used to specify the name of the programming language, for example `C`.
- `producer` is used to specify the name of the tool that produced the witness automaton, for example `CPAchecker 1.6.8`.
- `specification` is used to provide a textual representation of the specification of the verification task. The format of this representation is user-defined. In SV-COMP [12], the text `CHECK ( init(main()), LTL(G ! call(__VERIFIER_error())))` is used to represent the specification from Figure 2.
- `programfile` is used to record the URI or file-system path to the source code, e.g., `loop-acceleration/multivar_true-unreach-call1_true-termination.i`. This key is intended for documentation purposes; a validator is not required to be able to access the specified file location, because the source code is explicitly provided to the validator as input. Hence, the validity of the witness must not depend on the availability of the source code at the location specified by the value of this key in the execution environment of the validation.
- `programhash` is used to record the SHA-256 hash value of the verified program.
- `architecture` is used to provide a textual representation of the machine architecture assumed for the verification task. This textual representation is user-defined. We propose to use the identifiers `32-bit` and `64-bit` to distinguish between 32-bit systems and 64-bit systems.
- `creationtime` is used to specify the date and time the witness was created in ISO 8601 format. The date must contain the year, the month, and the day, separated by dashes (“`-`”). The date is separated from the time using the capital letter “`T`”. The time must be given in hours, minutes, and seconds, separated by colons (“`:`”). If the timestamp is in UTC time, it ends with a “`Z`”. If the timestamp is not given in UTC time, a positive (“`+`”) or negative (“`-`”) time offset consisting of hours and minutes separated by a colon (“`:`”) can be appended. Example: `2016-12-24T13:15:32+02:00`.

We require that values for all keys listed above are provided. The value of the `attr.type` attribute of all `key` elements corresponding to these keys is `string`.

Keys for `node` Elements. The following keys are defined for `node` elements, which represent control states in witness automata (cf. Table 4):

Table 4. Keys for node Elements Allowed in Witnesses

Key	Violation witness	Correctness witness
entry	✓	✓
sink	✓	
violation	✓	
invariant		✓
invariant.scope		✓
cyclehead	✓	

— entry is used to mark a node as an entry node. An entry node represents the initial control state of the witness automaton. We require that exactly one initial control state is defined per document. The attr.type attribute of this key is Boolean. The default value is false.
— sink is used to mark a node as a sink node. A sink node represents a sink control state in the automaton. Sink states are not allowed in correctness-witness automata (Tables 1 and 4). The attr.type attribute of this key is Boolean. The default value is false.
— violation is used to mark a control state as a violation state, i.e., as a state that represents a specification violation. Violation control states are not allowed in the syntax for correctness witnesses, because all control states are implicitly accepting states (Tables 1 and 4). The attr.type attribute of this key is Boolean. The default value is false.
— invariant is used to specify an invariant for a control state. The value of a data element with this key must be an expression that evaluates to a value of the equivalent of a Boolean type in the programming language of the verification task, e.g., for C programs, a C expression that evaluates to a value of the C type int (used as Boolean). The expression may consist of conjunctions or disjunctions, but not function calls. Local variables that have the same name as global variables or local variables of other functions can be qualified by using a data element with the invariant.scope key. Invariants are not allowed in violation-witness automata (Tables 1 and 4). The attr.type attribute of this key is string. All variables used in the expression must appear in the program source code. If a control state does not have a data element with this key, a consumer shall consider the state invariant to be *true*.
— invariant.scope is used to qualify variables with ambiguous names in a state invariant by specifying a function name: The witness consumer must map the variables in the given invariant to the variables in the source code. Due to scopes in many programming languages, such as C, there may be ambiguously named variables in different scopes. The consumer first has to look for a variable with a matching name in the scope of the function with the name specified via a data element with the invariant.scope key before checking the global scope. This key always applies to the invariant as a whole, i.e., it is not possible to specify an invariant over local variables of different functions. In existing implementations, there is currently no support for different variables with the same name within different scopes of the same function. Invariant scopes are not allowed in violation-witness automata (Tables 1 and 4). The attr.type attribute of this key is string.
— cyclehead is used to mark a state that connects stem and loop in a violation witness for termination, i.e., it marks the separation of *stem* and *loop* of a non-termination *lasso* [82]. A state with this annotation should be reachable from every non-sink state in the loop. At least one such state is required in a violation witness for termination properties. In reachability witnesses, this annotation is not allowed. The attr.type attribute of this key is Boolean. The default value is false.

Table 5. Keys for `edge` Elements Allowed in Witnesses

Key	Violation witness	Correctness witness
`assumption`	✓	✓
`assumption.scope`	✓	✓
`assumption.resultfunction`	✓	✓
`control`	✓	✓
`startline`	✓	✓
`endline`	✓	✓
`startoffset`	✓	✓
`endofset`	✓	✓
`enterLoopHead`	✓	✓
`enterFunction`	✓	✓
`returnFromFunction`	✓	✓
`threadId`	✓	✓
`createThread`	✓	✓

In general, it is not required to annotate a `node` element with `data` elements, except (a) that one `node` must be specified to represent the initial state of the automaton using a `data` element with the `entry` key, (b) that in a violation-witness automaton, there should be at least one control state that is marked as a violation state using a `data` element with the `violation` key, and (c) that in a violation witness for a termination specification, there should be at least one control state that uses a `data` element with the `cyclehead` key.

Keys for `edge` Elements. The following keys are defined for `edge` elements, which represent transitions in witness automata (cf. Table 5):

— `assumption` is used to specify a state-space guard for a transition. The value of a `data` element with this key must be an expression that evaluates to a value of the equivalent of a Boolean type in the programming language of the verification task, e.g., for C programs, a C expression that evaluates to a value of the C type `int` (used as Boolean). The expression may consist of conjunctions or disjunctions, but not function calls. Local variables that have the same name as global variables or local variables of other functions can be qualified by using a `data` element with the `assumption.scope` key. All variables used in the expression must appear in the program source code, with the exception of the variable `\result`, which represents the return value of a function identified by the `data` element with the key `assumption.resultfunction` after a function-return operation on a CFA edge matched by this transition. If the `\result` variable is used, the name of the corresponding function must be provided using a `data` element with the `assumption.resultfunction` key. If a transition does not have a `data` element with the `assumption` key, a consumer shall assume that the state-space guard of this transition is *true*. In correctness witnesses, for each state and each source-code guard, the disjunction of all state-space guards leaving that state via a transition matched by that source-code guard must be *true*, i.e., while state-space guards can be used to split the state space in correctness witnesses, they may not be used to restrict it (Table 1). The `attr.type` attribute of this key is `string`.

— `assumption.scope` is used to qualify variables with ambiguous names in a state-space guard by specifying a function name: The witness consumer must map the variables in the given invariant to the variables in the source code. Due to scopes in many programming languages, such as C, there may ambiguously named variables in different scopes. The consumer first has to look for a variable with a matching name in the scope of the function with the name

specified via a `data` element with the `assumption.scope` key before checking the global scope. This key always applies to the state-space guard as a whole, i.e., it is not possible to specify a state-space guard over local variables of different functions. In existing implementations, there is currently no support for different variables with the same name within different scopes of the same function. The `attr.type` attribute of this key is `string`.

- `assumption.resultfunction` is used to specify the function of the `\result` variable used in a state-space guard of the same transition, meaning that `\result` represents the return value of the given function. This key applies to the state-space guard as a whole, it is, therefore, not possible to refer to multiple function-return values within the same transition. If the `\result` variable is used, a `data` element with this key must be used in the same transition, otherwise, it is superfluous. The `attr.type` attribute of this key is `string`.
- `control` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to assume operations of the CFA. Valid values for `data` elements with this key are `condition-true` and `condition-false`, where `condition-true` specifies the branch where the assumed condition evaluates to *true*, i.e., the `then` branch, and `condition-false` specifies the branch where the assumed condition evaluates to *false*, i.e., the `else` branch. The `attr.type` attribute of this key is `string`.
- `startline` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations on specific lines in the source code. Any line number of the source code is a valid value for `data` elements with this key. A `startline` refers to the line number on which an operation of a CFA edge begins. The `attr.type` attribute of this key is `int`.
- `endline` is similar to the `startline` key, except that it refers to the line number on which an operation of a CFA edge ends.
- `startoffset` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations between specific character offsets in the source code, where the term character offset refers to the total number of characters from the beginning of a source-code file up to the beginning of some intended statement or expression. Any character offset between the beginning and end of a source-code file is a valid value for `data` elements with this key. While on the one hand, usage of `data` elements with this key allows the witness to convey very precise location information, on the other hand, this information is susceptible to even minor changes in the source code. If this is not desired, usage of `data` elements with this key should be omitted by the producer, or, if that is not an option, it can be removed during a post-processing step, provided that enough other source-code guards are present to make matching the witness against the source code feasible. A third option would be to recompute the offset values for the changed source code using a diff tool. The `attr.type` attribute of this key is `int`.
- `endoffset` is similar to the `startoffset` key, except that it refers to the character offset at the end of an operation.
- `enterLoopHead` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations on CFA edges where the successor is a loop head. For our format specification, any CFA node that (1) is part of a loop in the CFA, (2) has an entering CFA edge where the predecessor node is not in the loop, and (3) has a leaving CFA edge where the successor node is not in the loop, qualifies as a loop head. Note, however, that depending on the programming language of the verification task, the loop head may be ambiguous. For example, in C it is possible to use `goto` statements to construct arbitrarily complex loops with many CFA nodes that match the definition above. Conversely, there could also be loops without any loop head matching this definition, in

which case the key may not be used. The `attr.type` attribute of this key is `Boolean` and its default value is `false`.

- `enterFunction` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to function-call operations where the name of the called function matches the specified value. A witness consumer may also use this key to track a stack of function calls and use this information to qualify ambiguously named variables in state-space guards or state invariants in the absence of explicitly specified scopes via the `assumption.scope` or `invariant.scope` keys. The `attr.type` attribute of this key is `string`.
- `returnFromFunction` is the counterpart of the `enterFunction` key, i.e., it is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to function-return operations where the name of the function that is being returned from matches the specified value. Analogously to `enterFunction`, a witness consumer may use this key to track a stack of function calls. The `attr.type` attribute of this key is `string`.
- `threadId` is used in the analysis of concurrent programs[11] as part of the source-code guard of a transition and represents the currently active thread for the transition. The value of `data` elements with this key must uniquely identify an active (i.e., created but not yet destroyed) thread within each run through the automaton, meaning that if two different threads share the same identifier, they must either (a) be on different automaton runs or (b) at each step of each automaton run at most one of them may be active. If a transition has `data` elements where one specifies a `threadId` and another one uses the `createThread` key, the `threadId` refers to the thread that creates the new thread, not the created thread. The `attr.type` attribute of this key is `string`.
- `createThread` is used in the analysis of concurrent programs as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations where a new thread is created. The value of `data` elements with this key is an identifier for the new thread. Any string may be used as an identifier, provided that it uniquely identifies an active thread in each automaton run. The initial function of the created thread must be provided in a subsequent automaton transition using the `enterFunction` key, except for the main thread of the program, where the same (initial) transition may be used because, at that point, no other thread exists yet. Subsequently, a thread is assumed to be terminated once its callstack is empty again, which is achieved by using a corresponding `returnFromFunction` value. The `attr.type` attribute of this key is `string`.

In general, it is not required to annotate an `edge` element with `data` elements, but in practice, there is rarely any value in having a completely unrestricted transition in a protocol automaton. Note that the *o/w*-transitions that we defined in Section 3.1.2 are implicit, i.e., they do not appear in the exchange format as explicit `edge` elements but are automatically synthesized by the consumer.

The format specification, including a list of keys, is maintained in a GitHub project.[12] For termination witnesses, the project contains a dedicated section.[13] An open-source witness linter for checking the well-formedness of a witness is also available[14] and has been used in SV-COMP 2021 [18].

[11] For more details on witnesses for concurrent programs, we refer the reader to the literature [29].
[12] https://github.com/sosy-lab/sv-witnesses.
[13] https://github.com/sosy-lab/sv-witnesses/tree/svcomp22/termination.
[14] https://github.com/sosy-lab/sv-witnesses/tree/svcomp22/lint.

6 EXPERIMENTAL EVALUATION

To demonstrate the applicability of our approach, we performed a large number of experiments. The experimental work flow consists of running (1) a verifier, which produces a verification witness for the obtained result, and (2) a validator, which uses the verification witness to validate the result that the verifier obtained.

6.1 Experiment Goals

In the previous section, we defined an exchange format for machine-readable witnesses, in order to enable different verifiers to document their verification results in such a way that other tools can work with those verification results. Next, we perform an experimental study to support the following claims:

Claim 1 (Consistency within the Same Framework): Most of the witnesses produced by a verifier based on a certain framework can be validated by a validator based on the same framework. If the claim does not hold, then there is an inconsistency in the communication of the verification facts via the witnesses.

Claim 2 (Validation across Frameworks): The witnesses produced by a verifier based on one framework can be understood by a witness validator of a different framework.

Claim 3 (Effectiveness and Efficiency of Validation Depends on Witness Contents): There are verification tasks for which a verifier can produce witnesses such that the validation uses less resources to validate the result based on the witness, than the verifier used to solve the verification task.

We evaluate these claims separately for violation witnesses and correctness witnesses. Furthermore, we distinguish between five different categories (which correspond to five different specifications) of verification tasks, because not all verifiers and validators support all categories (see Table 6).

6.2 Benchmark Set

Our benchmark set consists of all 10,521 verification tasks from all categories of SV-COMP 2019 [13], for 3,740 of which there is a known specification violation, i.e., we expect a verifier to find a bug and document it with a violation witness, whereas no violation is known for the other 6,781, i.e., we expect a verifier to find a correctness proof and document it with a correctness witness.

We used CPAchecker and UAutomizer as verifiers for all of these tasks, but due to technical limitations not every validator supports all features required to analyze violation witnesses and correctness witnesses for each category of tasks.

Table 6 depicts which task category is supported by which validator.[15] In SV-COMP 2019, no validator existed that supports the validation of correctness witnesses for the categories Concurrency and Termination.

6.3 Experimental Setup

Our experiments were conducted on machines with a 3.4 GHz 8-core CPU (Intel Xeon `E3-1230 v5`) with 33 GB of RAM. The operating system was Ubuntu `18.04` (64-bit), using Linux `4.15` and OpenJDK `1.8`. Each run for a single verification or validation task was limited to two CPU cores, a CPU run time of 15 min, and a memory usage of 15 GB. The benchmarks were executed using BenchExec [43] in version `1.17`.

[15]The benchmark definitions for all validators can be found at: https://github.com/sosy-lab/sv-comp/tree/svcomp19/benchmark-defs.

Table 6. Categories Supported by Witness Validators

Category	Witness type	CPAchecker	CPA-witness2test	FShell-witness2test	UAutomizer
Concurrency	Violation	✓			
	Correctness				
MemSafety	Violation	✓	✓	✓	✓
	Correctness				✓
Overflows	Violation	✓	✓	✓	✓
	Correctness				✓
ReachSafety	Violation	✓	✓	✓	✓
	Correctness	✓			✓
Termination	Violation	✓			✓
	Correctness				

6.3.1 Verifiers. We used two verifiers, CPAchecker and UAutomizer. CPAchecker was used in version `cpachecker-1.7-witnesses-tosem-20181130` (revision 29 913 from the trunk). We configured it to use MathSAT5 as an SMT solver. As in our preliminary work on correctness witnesses [23], we use *k*-induction with auxiliary-invariant generation for the tasks from category ReachSafety, as defined in configuration `svcomp18--kInduction`. For the other categories, which were not part of our preliminary evaluation, we use the corresponding analyses from the CPA-Seq submission for SVCOMP 2019, as defined in configuration `svcomp19--concurrency` for category Concurrency, `svcomp19--memorysafety` for category MemSafety, `svcomp19--overflow` for category Overflows, and `svcomp19--termination` for category Termination. UAutomizer was used in its SVCOMP 2019 version (`0.1.24-91b1670e`) with Z3 as an SMT solver.

6.3.2 Validators. The categories supported by the validators are given in Table 6.

For the static validator based on CPAchecker, the same version as for the verifier was used with the configuration `witnessValidation`. This configuration performs violation-witness-based result validation by using CPAchecker's framework for configurable program analysis (cf. Section 4.1.1) to compose predicate analysis and explicit-state model checking into a combined analysis, as described in Section 4.2.2. To perform correctness-witness-based result validation, this configuration uses *k*-induction, where instead of synthesizing invariants itself like a verifier would, the validator uses only auxiliary invariants from the set of confirmed candidate invariants from the witness, as described in Section 4.3.2.

For the static validator based on UAutomizer, the same version as for the verifier was used and configured to perform witness-based result validation, which handles violation witnesses as described in Section 4.2.2 and correctness witnesses as described in Section 4.3.2.

CPA-witness2test was used in the same version of the CPAchecker framework as the CPAchecker-based verifier and was configured to perform the dynamic result validation described in Section 4.2.2.

FShell-witness2test was used in revision `c15c8acb` from its repository[16] and was configured to perform the dynamic result validation described in Section 4.2.2.

6.3.3 Presentation. All reported times (CPU time) are rounded to two significant digits. If the validation of a witness exceeds its resource limits before confirming the witness, then the validation result is counted as unconfirmed. The HTML tables in the reproduction package and on the supplementary web page (see Section 8) are generated with the table generator from BenchExec.

[16] https://github.com/tautschnig/fshell-w2t.

Table 7. Confirmed and Unconfirmed Violation Results in the Category Concurrency

Validator	CPAchecker	
Producer	CPAchecker	Automizer
Confirmation rates:		
Produced	772	247
Confirmed	771	4
Unconfirmed	1	243
Confirmation rate	100%	1.6%

Table 8. Confirmed and Unconfirmed Violation Results in the Category MemSafety

Validator	CPAchecker		CPA-witness2test		FShell-witness2test		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:								
Produced	107	67	107	67	107	67	107	67
Confirmed	106	54	17	15	29	1	27	43
Unconfirmed	1	13	90	52	78	66	80	24
Confirmation rate	99%	81%	16%	22%	27%	1.5%	25%	64%

6.4 Results

6.4.1 Violation Witnesses. Table 6 shows that for violation witnesses, all four validators can be used, which the execution-based validators CPA-witness2test and FShell-witness2test support all categories except Concurrency and Termination, CPAchecker supports all categories, and UAutomizer supports all categories except Concurrency.

Claim 1: Consistency within the Same Framework. Our first experiment for violation witnesses represents a study showing that we were able to implement a witness exchange format for violation witnesses for C programs for CPAchecker and UAutomizer, where both can take the roles of a verifier (producing witnesses) and also, for categories where the corresponding tool supports validation, of a witness validator for their own witnesses. Additionally, because CPA-witness2test is also based on the CPAchecker framework, we expect CPA-witness2test to also be able to validate witnesses produced by the CPAchecker verifier.

Category Concurrency. The first column of Table 7 shows that in the category Concurrency, CPAchecker confirmed 771 of 772 witnesses produced by CPAchecker, so that the confirmation rate for results produced by the same framework the validator is based on is almost 100 %.

Category MemSafety. The first, third, and last columns of Table 8 show that in category MemSafety, CPAchecker confirmed 106 of 107 witnesses produced by CPAchecker, that CPA-witness2test confirmed 17 of 107 witnesses produced by CPAchecker, and that Automizer confirmed 43 of 67 witnesses produced by Automizer, so that the confirmation rates for results produced by the same framework the validator is based on are 99 %, 16 %, and 64 %, respectively.

Category Overflows. The first, third, and last columns of Table 9 show that in the category Overflows, CPAchecker confirmed 164 of 165 witnesses produced by CPAchecker, that CPA-witness2test confirmed 149 of 165 witnesses produced by CPAchecker, and that Automizer confirmed 163 of 163 witnesses produced by Automizer, so that the confirmation rates for results produced by the same framework the validator is based on are 99 %, 90 %, and 100 %, respectively.

Category ReachSafety. The first, third, and last columns of Table 10 show that in category ReachSafety, CPAchecker confirmed 920 of 964 witnesses produced by CPAchecker, that CPA-witness2test

Table 9. Confirmed and Unconfirmed Violation Results in the Category Overflows

Validator	CPAchecker		CPA-witness2test		FShell-witness2test		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:								
Produced	165	163	165	163	165	163	165	163
Confirmed	164	161	149	9	121	24	160	163
Unconfirmed	1	2	16	154	44	139	11	0
Confirmation rate	99%	99%	90%	5.5%	73%	15%	97%	100%

Table 10. Confirmed and Unconfirmed Violation Results in the Category ReachSafety

Validator	CPAchecker		CPA-witness2test		FShell-witness2test		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:								
Produced	964	491	964	491	964	491	964	491
Confirmed	920	271	698	218	554	184	634	438
Unconfirmed	44	220	266	273	410	307	330	53
Confirmation rate	95%	55%	72%	44%	57%	37%	66%	89%

Table 11. Confirmed and Unconfirmed Violation Results in the Category Termination

Validator	CPAchecker		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:				
Produced	575	558	575	558
Confirmed	568	432	557	548
Unconfirmed	7	126	707	10
Confirmation rate	99%	77%	97%	98%

confirmed 698 of 964 witnesses produced by CPAchecker, and that UAutomizer confirmed 438 of 491 witnesses produced by CPAchecker, so that the confirmation rates for results produced by the same framework the validator is based on are 95 %, 72 %, and 89 %, respectively.

Category Termination. The first and last columns of Table 11 show that in category Termination, CPAchecker confirmed 568 of 575 witnesses produced by CPAchecker and that Automizer confirmed 548 of 558 witnesses produced by Automizer, so that the confirmation rates for results produced by the same framework the validator is based on are 99 % and 98 %, respectively.

We see that overall, we often achieve high confirmation rates if we apply validators to verification results produced by verifiers that are based on the same frameworks, although there is still some room for improvement regarding the validation of violation results by Automizer in category MemSafety (64 %). We attribute the lowest and third-lowest confirmation rates in this experiment, namely the 16 % achieved by CPA-witness2test for validating the results of CPAchecker in category MemSafety and the 66 % achieved by CPA-witness2test for validating the results of CPAchecker in category ReachSafety, to the fact that execution-based validators, in general, require very precise witnesses with concrete variable assignments for all input variables and otherwise fail, whereas model-checking-based validators, such as CPAchecker and Automizer, are often able to compute missing variable assignments during validation [26].

Claim 2: Validation across Frameworks. Our second experiment represents a study showing that we were able to communicate violation witnesses across frameworks, where verification results

produced by the CPAchecker-based verifier are validated by the Automizer-based validator and vice versa, where verification results produced by the CPAchecker-based verifier and the Automizer-based verifier are validated by the dynamic validator FShell-witness2test, and where verification results produced by the Automizer-based verifier are validated by the dynamic validator CPA-witness2test.

Category Concurrency. The last column of Table 7 shows that this claim does not hold in the category Concurrency: We see that CPAchecker confirmed only 1.6 % of the verification results produced by Automizer. We attribute this to the fact that Automizer only recently added support for verifying tasks in this category and has not yet fully implemented all features required to produce witnesses that can easily be validated. While this shows that there still remains work to be done to better support this combination, we chose to include the results for this part of the experiment for completeness and to accurately report the state of the art regarding available implementations.

Category MemSafety. Table 8 shows that in the category MemSafety, while CPAchecker confirmed 81 % of the verification results produced by Automizer, Automizer confirmed only 25 % of the verification results produced by CPAchecker, which matches an observation from the previous experiment, namely that the support for the validation of violation results is still prototypical for the Automizer-based validator in category MemSafety. The results of the model-checking-based validators in the remaining categories are more promising, although it is expected that result validation across frameworks is more difficult than within the same framework. Table 8 also shows that CPA-witness2test confirmed 22 % of the verification results produced by Automizer, that FShell-witness2test confirmed 27 % of the verification results produced by CPAchecker, and that FShell-witness2test confirmed 1.5 % of the verification results produced by Automizer.

Category Overflows. Table 9 shows that in the category Overflows, CPAchecker confirmed 99 % of the verification results produced by Automizer, that CPA-witness2test confirmed 5.5 % of the verification results produced by Automizer, that FShell-witness2test confirmed 73 % of the verification results produced by CPAchecker, that FShell-witness2test confirmed 15 % of the verification results produced by Automizer, and that Automizer confirmed 97 % of the verification results produced by CPAchecker.

Category ReachSafety. Table 10 shows that in the category ReachSafety, CPAchecker confirmed 55 % of the verification results produced by Automizer, that CPA-witness2test confirmed 44 % of the verification results produced by Automizer, that FShell-witness2test confirmed 57 % of the verification results produced by CPAchecker, that FShell-witness2test confirmed 37 % of the verification results produced by Automizer, and that Automizer confirmed 66 % of the verification results produced by CPAchecker.

Category Termination. Table 11 shows that in the category Termination, CPAchecker confirmed 77 % of the verification results produced by Automizer and that Automizer confirmed 97 % of the verification results produced by CPAchecker.

As observed in the previous experiment, the confirmation rates achieved by the execution-based validators CPA-witness2test and FShell-witness2test are mostly lower than those achieved by the model-checking-based validators CPAchecker and Automizer due to their requirement for more precise witnesses. For example, FShell-witness2test is only able to validate 1 of 67 results produced by Automizer in category MemSafety, CPA-witness2test is only able to validate 9 of 163 results produced by Automizer in category Overflows, and the confirmation rate of 37 % for FShell-witness2test validating the results of Automizer in category ReachSafety appears low if compared directly with the results achieved by the model-checking-based validators. On the other hand, FShell-witness2test is able to validate more results produced by CPAchecker than Automizer is able to validate in category MemSafety. Moreover, while there are generally fewer confirmations by the execution-based

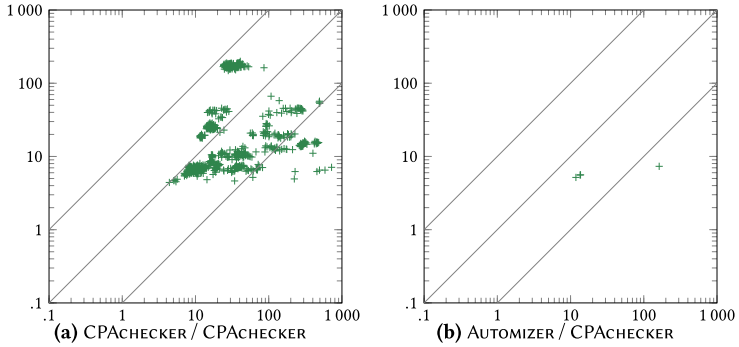


Fig. 14. Category Concurrency (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x-axis, CPU seconds for result validation on the y-axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c".

validators, these confirmations can be considered more valuable than the confirmations by model-checking-based validators in that they instill a higher confidence in the result and are bundled with easily debuggable executables for the verification result. For example, these numbers still show that the completely independent validator FShell-witness2test was able to synthesize executable binaries from 184 out of 491 witnesses produced by Automizer in category ReachSafety, execute them, and successfully replay the reported bugs, which gives a potential user not only a high confidence that these 184 bugs actually exist, but also provides observable, executable proofs for each confirmation.

Claim 3: Effectiveness and Efficiency of Validation Depends on Witness Contents. Our experiments also confirm that often, witness-based violation-result validation is faster than the corresponding preceding verification, although there are exceptions to this observation.

Category Concurrency. For example, Figure 14(a) shows that in the category Concurrency, using CPAchecker to validate verification results produced by CPAchecker is in most cases faster than the verification, but that there is also a small cluster of verification results where validation is almost ten times slower than the verification. For completeness, we also depicted in Figure 14(b) the scatter plot that compares the verification times of Automizer to the validation times of CPAchecker for Automizer's results. However, even though in this figure, validation is always faster than verification, we do not consider the low number of validated results significant enough to draw any general conclusions.

Category MemSafety. Figure 15(a) shows no significant time differences for category MemSafety between verifying a task with CPAchecker and validating the corresponding result with CPAchecker. On the other hand, Figure 15(b) shows that verification results produced by Automizer are always validated faster by CPAchecker than they were produced. The same, however, is not true for the inverse case. Figure 15(g) shows that in fact, verification results produced by CPAchecker are always validated slower by Automizer than they were produced, and Figure 15(h) shows only a few cases where validating verification results produced by Automizer are validated quicker by Automizer itself than they were produced. This matches our previous observation that the support for validating violation results is still prototypical for the Automizer-based validator. Figure 15(c)–(f), which are scatter plots for the verification results produced by CPAchecker and validated by CPA-witness2test, verification results produced by Automizer and validated by CPA-witness2test, verification results produced by CPAchecker and validated by FShell-witness2test, and verification results produced by Automizer and validated by FShell-witness2test, respectively, show that execution-based validation

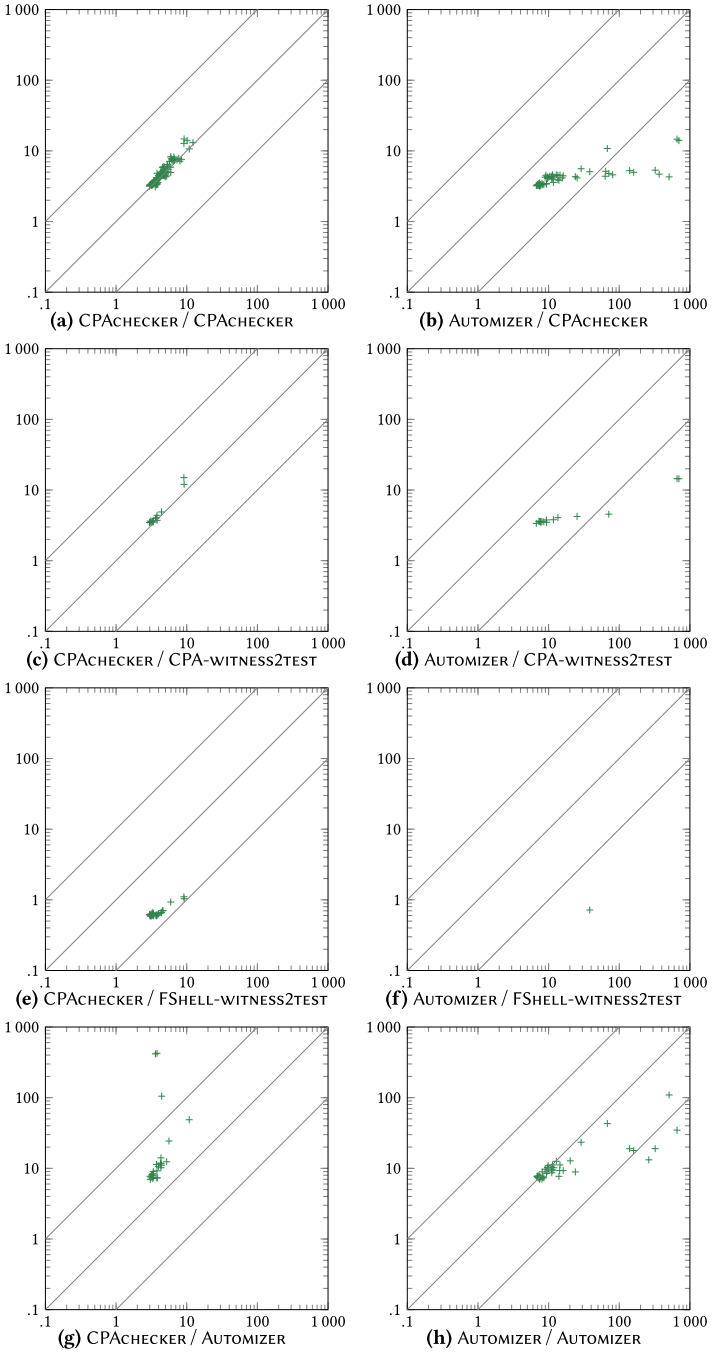


Fig. 15. Category MemSafety (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x-axis, CPU seconds for result validation on the y-axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c".

of results is mostly faster than verification although it must be noted that due to the low number of validations, this observation is not significant.

Category Overflows. Figure 16 shows for category Overflows almost no significant time differences for the model-checking-based validators between verifying a task and validating the corresponding verification result, which can be attributed to the fact that almost all tasks can already be verified in less than 10 s, i.e., very quickly, so that there is not much time to be gained by using a witness to reduce the search space during the validation. As for the category MemSafety, we again observe that the execution-based validators, are at least as quick and, in the case of FShell-witness2test, often even significantly faster than the corresponding verifications. However, the low number of confirmations again prohibits deriving a general claim from this observation.

Category ReachSafety. Figure 17 shows a somewhat clearer picture for category ReachSafety: In Figure 17(a) we can see that except for an insignificant amount of outliers, validating a verification result produced by CPAchecker with CPAchecker is at least as fast as producing that verification result, and that this effect appears to scale well, because even for many tasks where the verification took more than 100 s, validation took only less than a tenth of that time. In Figure 17(b) we observe the same effect for verification results produced by Automizer and validated by CPAchecker, although not as pronounced as in the previous figure. Figure 17(c)–(f), which depict the verification results produced by CPAchecker and validated by CPA-witness2test, verification results produced by Automizer and validated by CPA-witness2test, verification results produced by CPAchecker and validated by FShell-witness2test, and verification results produced by Automizer and validated by FShell-witness2test, respectively, show that execution-based validation is usually significantly faster than verification, and often also faster than model-checking-based validation, even though fewer results can successfully be validated, which is particularly visible in Figure 17(e) and (f), which compare the validation times of FShell-witness2test to the corresponding verification times. Figure 17(g), on the other hand, shows that applying the Automizer-based validator to the verification results produced by CPAchecker, there are cases where validation is slower than, as fast as, or faster than verification, with no clearly discernible trend, which means that Automizer apparently often does not profit from the reduced search space provided by the witnesses of CPAchecker and its validation times are more dependent on its own engine than on the witnesses, whereas Figure 17(h) shows that Automizer can profit from its own witnesses, because the Automizer-based validator often validates a verification result in less time than Automizer took to produce it.

Category Termination. Lastly, for the category Termination, Figure 18(a) and (d) show that both the CPAchecker-based validator and the Automizer-based validator profit from witnesses for verification results produced by their own respective frameworks and often validate these results in less time than it took to produce them, whereas there is no apparent performance improvement visible in Figure 18(b) and (c), which compare the validation times of CPAchecker for the results produced by Automizer and the validation times of Automizer for the results produced by CPAchecker, respectively, to the corresponding verification times.

Summary. We observed that validation can be significantly faster than the preceding verification, but this effect is generally not guaranteed. In category ReachSafety, which is the largest of all five examined categories, we observe this effect even across verification frameworks. While these results are already promising, we interpret them as an indicator that more effort should be spent on improving witness-based validation of violation results, especially in the categories Concurrency, MemSafety, Overflows, and Termination, to achieve similar performance benefits as in category ReachSafety.

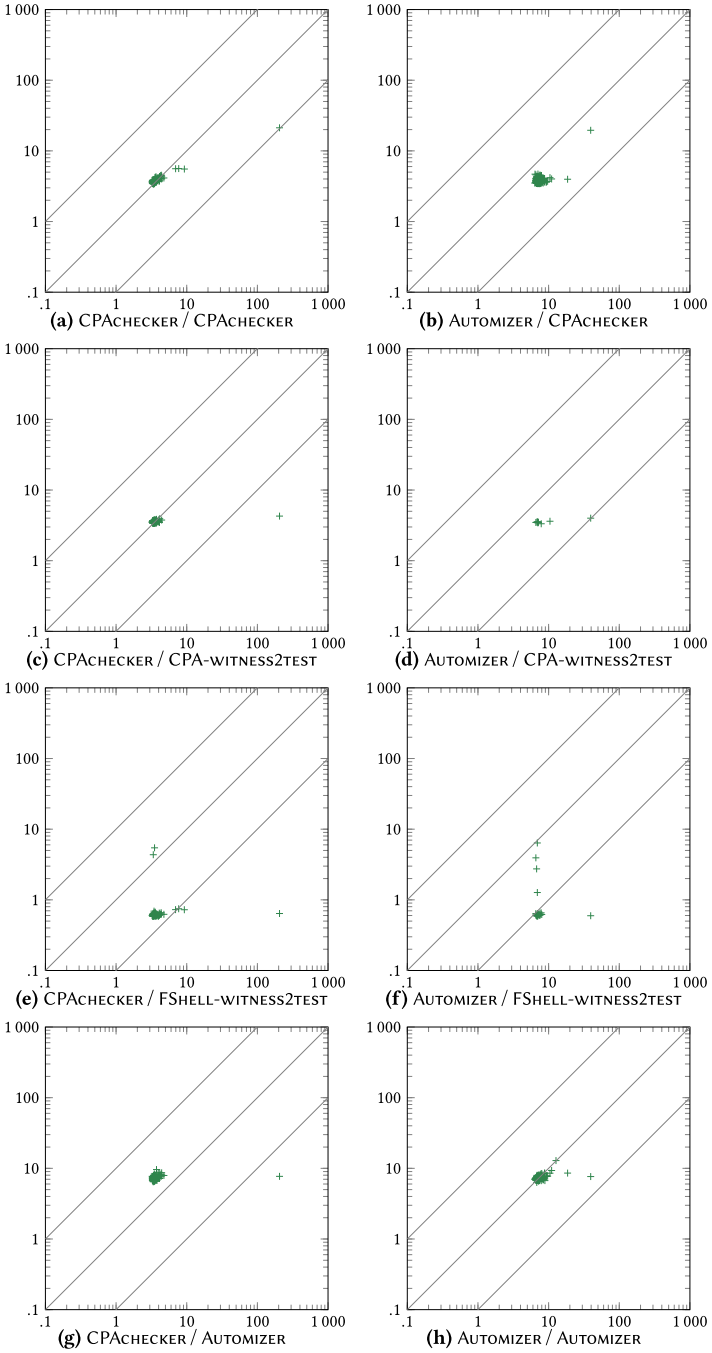


Fig. 16. Category Overflows (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x-axis, CPU seconds for result validation on the y-axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c".

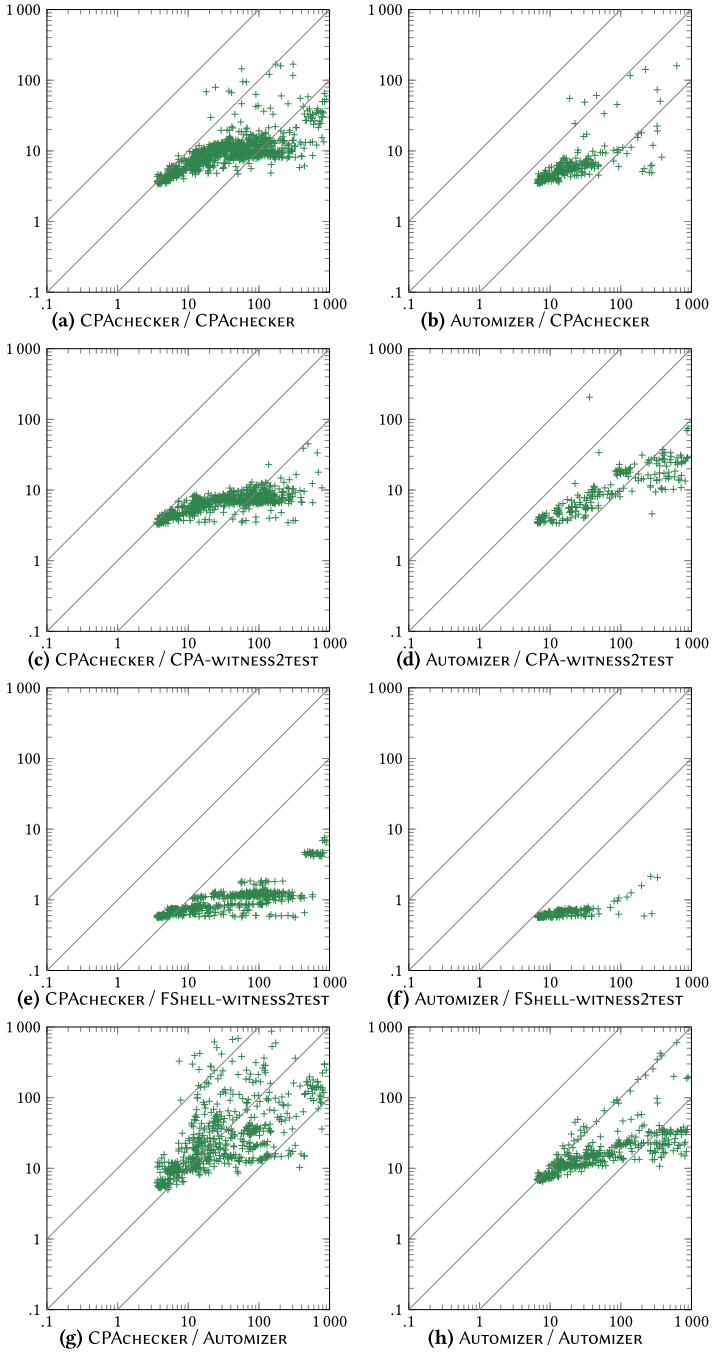


Fig. 17. Category ReachSafety (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x-axis, CPU seconds for result validation on the y-axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c".

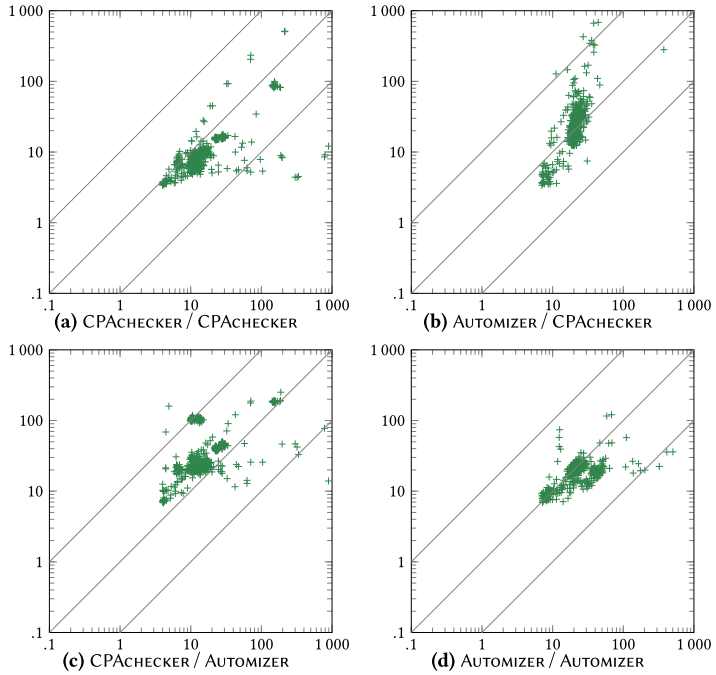


Fig. 18. Category Termination (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the *x*-axis, CPU seconds for result validation on the *y*-axis; a caption "*p*/*c*" abbreviates "witnesses produced by *p* that are confirmed by *c*".

6.4.2 Correctness Witnesses. Table 6 shows that for correctness witnesses, only CPAchecker and UAutomizer can be used as validators, and that UAutomizer supports categories MemSafety, Overflows, and ReachSafety, whereas CPAchecker supports only category ReachSafety.

Claim 1: Consistency within the Same Framework. Our first experiment for correctness witnesses represents a study showing that we were able to implement a witness exchange format for correctness witnesses for C programs for CPAchecker and UAutomizer, where both can take the roles of a verifier (producing witnesses) and also, if supported, a witness validator for their own witnesses. The last columns of Tables 12 and 13 show that Automizer confirmed 108 of 108 witnesses produced by Automizer in category MemSafety and 144 of 144 witnesses produced by Automizer in category Overflows, so that the confirmation rates of its own witnesses are 100 % in both cases. The first and last columns of Table 14 show that CPAchecker confirmed 2,130 of 2,642 witnesses produced by CPAchecker, and that Automizer confirmed 2,694 of 2,749 witnesses produced by Automizer, so that the confirmation rates for their own witnesses are 81 % and 98 %, respectively. Furthermore, for the rejected witnesses, UAutomizer detects incorrect invariants in 14 of its own witnesses, and CPAchecker refutes none of its own witnesses.[17]

[17]It may be interesting to developers of other verifiers to learn that when the development of the CPAchecker-based correctness-witness export and validation started, there were a lot more incorrect invariants, which were caused by several actual bugs in other components of the framework that the CPAchecker team had been unaware of. In addition to the other benefits, implementing correctness-witness validation can therefore also be a way to improve the overall quality of a verifier.

Table 12. Confirmed and Unconfirmed Correctness Results in the Category MemSafety

Validator	Automizer	
Producer	CPAchecker	Automizer
Confirmation rates:		
Produced	118	108
Confirmed	52	108
Unconfirmed	66	0
Confirmation rate	44%	100%

Table 13. Confirmed and Unconfirmed Correctness Results in the Category Overflows

Validator	Automizer	
Producer	CPAchecker	Automizer
Confirmation rates:		
Produced	130	144
Confirmed	119	144
Unconfirmed	11	0
Confirmation rate	92%	100%

Table 14. Confirmed and Unconfirmed Correctness Results in the Category ReachSafety

Validator	CPAchecker		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:				
Produced	2 642	2 749	2 642	2 749
Confirmed	2 130	1 297	1 827	2 694
Unconfirmed	512	1 452	815	55
Confirmation rate	81%	47%	69%	98%

Claim 2: Validation across Frameworks. Our second experiment represents a study showing that we were able to communicate correctness witnesses across frameworks, where verification results produced by the CPAchecker-based verifier are validated by the Automizer-based validator and vice versa. Tables 12 and 13 show that Automizer confirmed 44 % of the verification results produced by CPAchecker in category MemSafety and 92 % of the verification results produced by CPAchecker in category Overflows. Table 14 shows that in the category ReachSafety, CPAchecker confirmed 47 % of the verification results produced by Automizer, and that Automizer confirmed 69 % of the verification results produced by CPAchecker. Except for category Overflows, these numbers are not yet as favorable as those where the tools validate their own witnesses. We analyzed the unconfirmed results and found different causes for both cases: (1) CPAchecker did not detect any incorrect invariants in the witnesses produced by Automizer, and there are often too few invariants present in those witnesses for the k-induction-algorithm to succeed within the time limit. This means that CPAchecker mostly does not dispute the witnesses of Automizer, but it cannot confirm them either. (2) Automizer is not always able to find the correct program location for an invariant. If Automizer maps an invariant to the wrong program location, and thus, the invariant does not hold there, then the witness is rejected. While there is still room for improvement to our implementations, in general, the witnesses were understood by the validators of other frameworks, and the rejections are

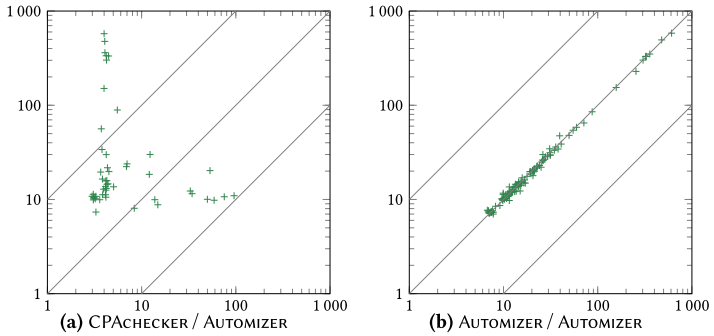


Fig. 19. Category MemSafety (correctness): Scatter plots for pairwise composition for witness-based correctness-result validation; CPU seconds for producing a witness on the x-axis, CPU seconds for result validation on the y-axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c".

mostly due to timeouts rather than due to wrong or miscommunicated invariants. Our experiment over the three categories MemSafety, Overflows, and ReachSafety, shows that for between 1,300 to 2,000 of 2,700 to 2,900 tasks verified by one verifier, a validator based on a different framework and different techniques not only agreed on the verdict but confirmed that no flaw was detected in the reasoning represented by the correctness witness, whereas previously, communicating such information between different tools was entirely impossible.

Claim 3: Effectiveness and Efficiency of Validation Depends on Witness Contents. Our experiments also confirm that the contents of the witnesses influences the difficulty of the validation, so that for a given verification task, one witness can lead to a quick validation, while a validation based on a different witness may require more resources or even fail to terminate at all. We first take a closer look at the differences in resource usage between verification and validation for a given task. Figure 21(a) shows that, especially for tasks that require more than 20 s of CPU time, CPAchecker produces three groups of witnesses, for which the validation is (a) about as fast as, (b) quicker than, and (c) slower than the preceding verification: The first group is explained by tasks for which few or even no auxiliary invariants are required by the k-induction technique. The second group is caused by tasks for which the witnesses contain useful invariants that allow the validator to quickly validate the task, while the verifier had to spend time on synthesizing the invariants. The third group represents tasks for which the witnesses contain significant amounts of invariants that turn out to be irrelevant, but the time spent by the validator to check them exceeds the time spent by the verifier to generate them. Figure 21(b) shows that many of the witnesses produced by Automizer that can be validated by CPAchecker are in most cases validated more quickly than they were produced. Figures 19(a), 20(a), and 21(c) are similar to Figure 21(a): there are cases for which the validation is faster than the verification and vice versa. But since in these three figures, validation and verification are performed by different tools, the differing characteristics of the two tools may outweigh the effects of the witnesses on validation speed: Automizer is often not faster at validating the invariants contained in the witnesses, and instead is often slower than CPAchecker for those of CPAchecker's witnesses that it can validate. It must also be noted that Figure 20(a) shows that most of the few tasks in category Overflows that can be verified and where a validator confirms the result, are solved in less than about 10 s, which suggests that in this category, comparing verification and validation times is not particularly meaningful. Figures 19(b), 20(b), and 21(d) show that for Automizer, there is no discernible difference between the CPU times required to produce a witness and to validate it.

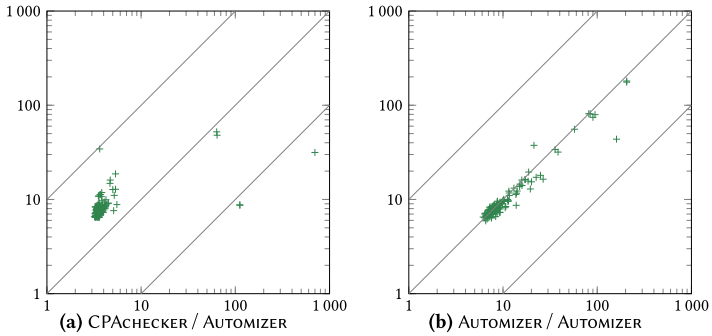


Fig. 20. Category Overflows (correctness): Scatter plots for pairwise composition for witness-based correctness-result validation; CPU seconds for producing a witness on the x-axis, CPU seconds for result validation on the y-axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c".

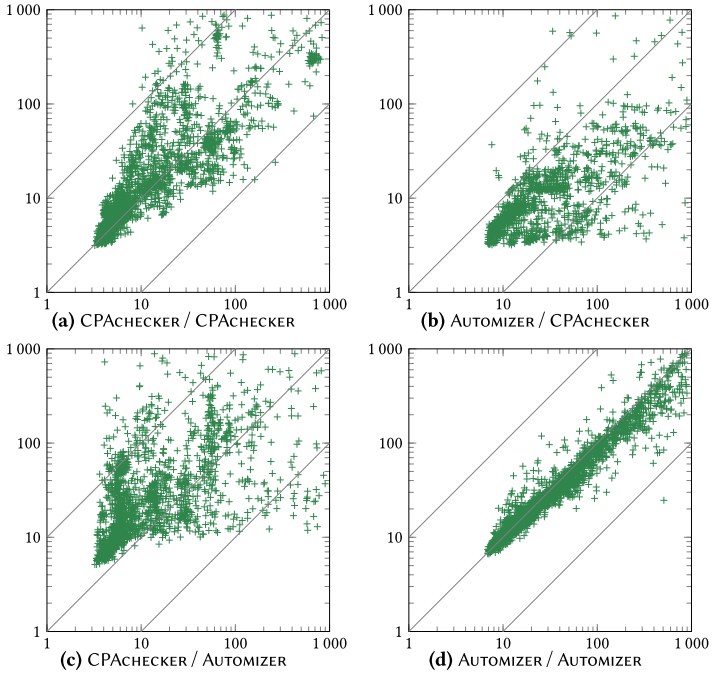


Fig. 21. Category ReachSafety (correctness): Scatter plots for pairwise composition for witness-based correctness-result validation; CPU seconds for producing a witness on the x-axis, CPU seconds for result validation on the y-axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c".

General Trend. In general, we could not observe a definite trend of speed-up overall validation runs using correctness witnesses. We attribute these results to the fact that it is not trivial to determine which invariants should be exported to the witness, because while exporting too much information unnecessarily complicates the validation, too few or too weak invariants may impede the feasibility of the validation. The fact that an invariant that suffices for one validator may not be sufficient for a different validator further complicates the decisions that drive the composition of invariants for a specific witness: suppose a verifier produces a witness that contains a 10-inductive invariant. A validator based on k-induction would likely be able to prove this invariant easily with $k = 10$,

Table 15. Examples of Verification Tasks for Which Correctness-witness-based Result Validation was Significantly Faster than the Verification Run That Produced the Correctness Witness

Program	Category	Verifier	Validator	Verifier CPU Time	Validator CPU Time
floppy_simpl4.cil.c	MemSafety	CPAchecker	Automizer	96 s	11 s
openbsd_cstrpbrk-alloca.i	MemSafety	Automizer	Automizer	38 s	34 s
Fibonacci02.c	Overflows	CPAchecker	Automizer	700 s	31 s
GopanReps-CAV2006-Fig1a.c.c	Overflows	Automizer	Automizer	160 s	44 s
minepump_spec2_product52.cil.c	ReachSafety	CPAchecker	CPAchecker	160 s	16 s
Problem15_label53.c	ReachSafety	CPAchecker	Automizer	720 s	98 s
test_locks_15.c	ReachSafety	Automizer	CPAchecker	300 s	6.6 s
tree.i	ReachSafety	Automizer	Automizer	510 s	25 s

whereas a validator based on some other technique would likely have to first synthesize auxiliary invariants. We can, however, provide examples of cases for various different types of verification tasks for which a speed-up exists[18]: Table 15 shows for each supported combination of category, verifier, and validator an example for which the validation of a correctness witness was faster than the verification run that produced the correctness witness. For combinations where the validator is based on the same framework as the verifier (i.e., CPAchecker/CPAchecker, Automizer/Automizer), the speedup cannot be dismissed as caused by differences in the underlying implementation; instead, the speedup suggests that there is value in the guidance provided by the correctness witness in these cases. Unsurprisingly, validation only benefits from invariants that are difficult to derive but can be proved easily. If, however, too much work is left to the validator, then the validation is slower than the verification, because, in addition to parsing the witness and matching its contents to the program, it also needs to synthesize its own invariants. Lastly, our implementations are based on generic model checkers and the potential for optimization towards validation is not yet utilized.

Summary. In conclusion, these experiments confirm that the contents of a correctness witness can be important for one of the validators (CPAchecker), while they do not seem to make a noticeable difference for the other validator (Automizer), which can confirm more results but in turn is slower than the validator based on CPAchecker. This choice of a tradeoff as to what constitutes an acceptable witness is one of the strengths of our flexible exchange format for correctness witnesses: Users may choose a quick but strict validator (rejects if invariant is too weak) or a slower but more tolerant one (constructs missing invariants), depending on their use case.

6.5 Tutorial

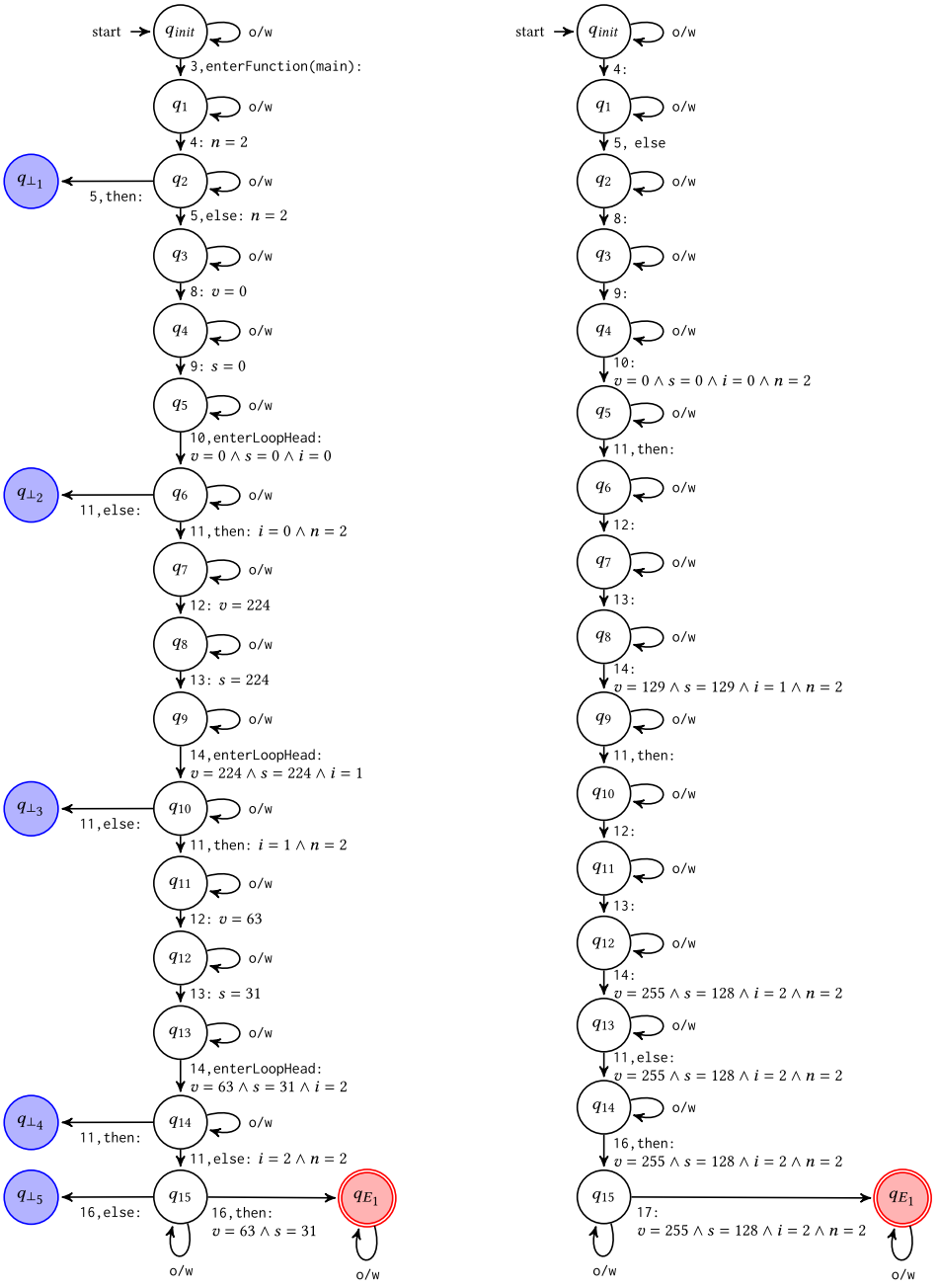
In order to collect initial experience with the process of witness-based result validation, we list here a selection of tool invocations to get started with. The verification task that we use in this tutorial consists of the C program linear-inequality-inv-b.c from Figure 9(a) and the specification unreach-call.prp for which the observer automaton is given in Figure 2.[19]

Verify a Program with a Given Specification.

For CPAchecker, the following command line produces a witness similar to Figure 22(a):

[18]We can pick the verification tasks from the bottom-right part of the scatter plots.

[19]Note that between SV-COMP 2020 and SV-COMP 2021, the `unreach-call` specification changed from reachability of function `__VERIFIER_error` to function `reach_error` (see [18], page 404). If the goal is to use the task exactly as presented here, it is advisable to use the tool versions from the reproduction package and the specification file from the SV-COMP 2019 release of the benchmark repository. In general, the latest versions of the tools can be used, as well as the latest version of the program and specification from the benchmark repository.



(a) Violation witness produced by CPAchecker **(b)** Violation witness produced by UAutomizer

Fig. 22. Violation witnesses produced by CPAchecker and UAutomizer for the verification task consisting of the C program linear-inequality-inv-b.c from Figure 9(a) and the specification unreach-call.prp from Figure 2.

```
scripts/cpa.sh \
  -spec sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

For UAutomizer, the following command line produces a witness similar to Figure 22(b):

```
./Ultimate.py \
  --spec sv-benchmarks/c/properties/unreach-call.prp \
  --file sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c \
  --architecture 32-bit
```

Validate the Result with the Produced Witness.

To attempt to validate a result using a witness `witness.graphml` with CPAchecker, execute the following command line:

```
scripts/cpa.sh \
  -witnessValidation \
  -witness witness.graphml \
  -spec sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

To attempt to validate a result using a witness `witness.graphml` with CPA-witness2test, execute the following command line:

```
scripts/cpa_witness2test.py \
  -witness witness.graphml \
  -spec sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

To attempt to validate a result using a witness `witness.graphml` with FShell-witness2test, execute the following command line:

```
./test-gen.sh \
  --graphml-witness ../CPAchecker/witness.graphml \
  -m32 \
  --propertyfile sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

To attempt to validate a result using a witness `witness.graphml` with UAutomizer, execute the following command line:

```
./Ultimate.py \
  --spec sv-benchmarks/c/properties/unreach-call.prp \
  --file sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c \
  --validate ../CPAchecker/witness.graphml \
  --architecture 32-bit
```

6.6 Validity

6.6.1 Benchmark Selection. For our benchmarking, we selected all full categories from the standard repository of software-verification tasks[20] without any restriction to subsets. Consequently, our experiments are performed over the largest openly available collection of verification tasks for the C programming language. For each category of the benchmark set, we show the results for all validators that support the category in 2019. While the main goal of this article is to show that the approach can work in practice, we have not further excluded those verification tasks from

[20] https://github.com/sosy-lab/sv-benchmarks.

the benchmark set for which the implementation is still insufficient, and instead show the results from the categories that are not yet well supported alongside those that are, to accurately represent the current state of the art in witness-based result validation, to pinpoint areas where further improvements are required, and to showcase the potential of witness-based result validation for areas where more mature implementations already exist.

Our knowledge about expected verification verdicts is based on the verdicts of the software-verification community.[20] In theory, it could be possible that an unconfirmed witness was not confirmed because the assumed bug does not exist, which is very unlikely because the benchmark sets is exposed to a lot of verification tools.

6.6.2 Verification Tools. Our implementations for producing and validating witnesses are based on several independent frameworks that use completely different technologies: CPAchecker implements a static approach to violation-witness-based result validation using a combination of predicate analysis and explicit-state model checking [39, 42], a static approach to correctness-witness-based result validation using *k*-induction [27], and a dynamic approach to violation-witness-based result validation (CPA-witness2test) that produces, runs, and checks executable tests [26]. UAutomizer uses an automata-based approach [84] to static witness-based result validation. FShell-witness2test is based on the test-vector format of FShell [89] and is independent from any model-checking framework. This means that while comparisons of speed between verification with one tool and validation with the other tool are only meaningful on a very coarse level, we can show that a wide variety of techniques can be used for witness-based result validation.

6.6.3 Reproducibility. All data presented, including verification tasks, witnesses, verifiers, and their configurations, are available on our supplementary web site (see Section 8). For controlling and measuring the computing resources used in our experiments, such as memory, CPU time, core, and memory assignment, we use the state-of-the-art benchmarking framework BenchExec [43], and thereby ensure that our results are accurate, reliable, and reproducible. To further improve the reproducibility of our experiments, we also selected the configurations of the verifiers that are used to produce verification results and witnesses with a focus on the stability of their results instead of on their general effectiveness. For example, for CPAchecker, a more effective configuration than the one used in our experiments was used in SV-COMP 2019 in the category ReachSafety, but since this configuration uses timers to dynamically switch between various analyses, its results are less stable than our choice of a single analysis, *k*-induction, for this category.

7 RELATED WORK

The exchange format for verification witnesses described in this article and the corresponding techniques for communicating verification witnesses across verification tools were introduced—initially only for violation witnesses—in 2015 [25]. In 2016, the format was extended to encompass correctness witnesses [23]. We give updated technical descriptions and evaluation results for existing witness-based result validators [23, 25, 26].[21] The flexibility, stability, and practical applicability of the exchange format are evidenced by the fact that it has already been successfully applied for several years now in the annual TACAS International Competition on Software Verification (SV-COMP) [10–12]. As a result, all competing verifiers now support the exchange format for verification witnesses and augment their verification results with it.

[21] MetaVal [44] and NitWit [122] are not included in our evaluation because they were developed after our evaluation was done in 2019.

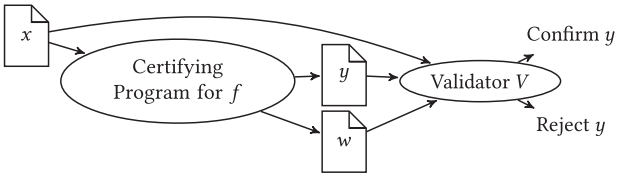


Fig. 23. A certifying program for a function f computes $y = f(x)$ and produces a witness w, which is then used to check the correctness of result y by a witness validator; adapted from [107].

7.1 Exchange Formats

Before the common exchange format became available, verification witnesses were used only based on proprietary formats within particular tools. For example, ESBMC was extended to reproduce errors via instrumented code [117], and CPACHECKER was used to validate previously computed error paths by interpreting them as witness automata that guide and restrict the state-space search [47]. There are other exchange formats as well: (1) The **Certification Problem Format** (**CPF**) [121] is used by the competition on termination [75] to store termination proofs for term-rewrite systems. (2) The DRAT [87] format is used in the SAT competitions [8] since several years in order to validate the correctness of proofs of unsatisfiability of a propositional formula using a witness validator for DRAT [128]. (3) The **Static Analysis Results Interchange Format** (**SARIF**) [115] is used to represent results from static analysis by some industrial tools, such as CODESONAR,[22] SWAMP,[23] and VISUAL STUDIO,[24] and which is mainly intended as input for visualization tools and for aggregating and embedding analysis results into bug-tracking or continuous-integration systems rather than for semantic analysis such as result validation.

7.2 Certifying Algorithms

The concept of *certifying algorithms* [107] is a solution for increasing trust in the results produced by potentially complex and error-prone computations. The paradigm of certifying algorithms demands that each algorithm provides, together with the computed output, a witness that in turn can be used to verify that the output is indeed a correct solution for the given input problem. Figure 23 illustrates this workflow for a certifying program for a function f, i.e., an implementation of a certifying algorithm: The program receives the input x, computes the result $y = f(x)$, and produces the witness w. The witness validator V receives the inputs x, y, and w, and leverages w to determine whether y is a correct result for $f(x)$. By certifying each individual result, the more difficult problem of proving the correctness of the certifying algorithm or its implementation is avoided. This concept applies to both violation witnesses and correctness witnesses: A violation witness is a certificate for a specification violation found by a verifier, whereas a correctness witness is a certificate for the proof found by a verifier. Both can be used by a validator to try to re-establish the verification result. The core advantages of using this approach are that to trust the verification result, it is not necessary to trust the producer of the witness, and that the result validator can re-establish the result independently. In fact, with verification witnesses, the tool that is used for the witness-based validation of a result can even work with a different abstract domain than the tool that produced the result and witness [111].

[22] https://www.grammatech.com/products/codesonar.
[23] https://github.com/mirswamp/deployment (see also https://continuousassurance.org/mir-swamp).
[24] https://visualstudio.microsoft.com.

7.3 Counterexamples

When a verifier detects a bug, it usually provides some form of counterexample [7, 12, 19, 57, 60, 61, 103]. On top of that, however, there is a growing demand for quick and automatic validation of program error paths to raise the confidence in automatically detected bug reports [12, 47, 68, 111], most importantly to reduce the number of false alarms. For example, an expensive, high-precision feasibility check can be used to filter out false alarms produced by an efficient, low-precision data-flow analysis within one instance of a verifier [68]. Experiments show that instead of repeating a full verification task from scratch, it is usually significantly faster to validate an existing verification result using a violation witness [25, 47]. However, without a unified exchange format for violation witnesses to export counterexamples and use them as input to another tool, full programs were synthesized from counterexamples and used as a medium [33, 38, 117]. For witness-based result validation, this approach is not useful, because the result and witness need to be checked against the original, unchanged program to ensure that no new error paths were introduced that did not exist originally. In the context of distributed high-performance computing, some exchange-formats for system traces exist, e.g., the MPI trace format [1], or the Open Trace Format [72, 102, 126], whose primary purpose is to keep a record of system events, such as messages that are exchanged between processes. These formats strongly focus on distributed systems with time-stamped events and are not applicable to our problem. Many applications for violation witnesses already exist [3, 22, 62, 71, 78, 79, 81, 98, 104, 131], and a common format that can be used to exchange witnesses across verification tools will stimulate further research in this direction, particularly on combinations of verification, debugging, and visualization techniques.

7.4 Test-case Generation

Verification counterexamples have been used to *generate test cases* for two decades now [19, 90, 91, 125]. Various automatic test-case generation techniques have been developed as extensions of this idea [80, 97, 119] and as combinations of counterexample-based test-case generation with other techniques, such as random testing [76, 106]. Test-goal automata are used to achieve specific coverage or to reach test goals by leading a program analysis towards specific program locations [35, 52]; conceptually, test-goal automata are simply a specific use case for the violation-witness automata that we present. Test cases from verification counterexamples have also been used to create debuggable executables [110, 117]. Two of the violation-witness-based result validators that we present [26] use this idea to validate verification results by synthesizing an executable from a verification task and a violation witness, and executing it to check if the reported error actually occurs. By using the common exchange format for witnesses, this technique can be applied to synthesize executables using the verification results of any tool that supports the common format. While other counterexample-based approaches for generating executable test cases [53, 67, 73, 105] are limited to concrete and tool-specific counterexamples, we do not require full counterexamples of any specific verifier; instead, our approach works on more flexible—and, thanks to our concept of witness refinement, potentially abstract—violation witnesses.

7.5 Correctness Certificates

There is a long history of *correctness certificates* for the purpose of increasing the trust in code that is generated from some form of formal description or model (e.g., [54, 58, 85, 94, 129]). While there are efforts to reduce the often inconveniently large size of these proofs [74], these correctness certificates are still complete proofs of functional correctness. While our exchange format can also be used as correctness certificate and to represent a full proof, this is not required: a correctness witness is more general, in that it can also be used as a *partial* proof of correctness [23], which can

be more concise than a full proof. Alan Turing suggested already in 1949 to annotate programs with assertions "from which the correctness of the whole program easily follows." [124]

7.6 Proof-carrying Code (PCC)

One application of correctness certificates has previously been explored in the context of ***proof-carrying code*** (**PCC**) [112]. PCC is a mechanism where an untrusted source supplies an executable program and a correctness certificate, both of which are therefore also untrusted initially. However, trust can be established by using a trusted validator to check the witness against the program and specification. Certifying model checkers can use the intermediate results of their verification procedure to compose full proofs and export them as proof certificates [111].

The exchange format for correctness witnesses allows the mechanism of proof-carrying code to be applied to real-world C programs and enables further verification tools to adopt the technique. Compared to previous publications on proof-carrying code, the main advantage that our exchange format and validation techniques provide is that we do not strictly require the witness to contain a full proof. We found that in practice, a complete proof for even short programs with simple specifications may become prohibitively large in size unless a considerable amount of additional effort is spent on simplifying formulas. Especially for more complex verification tasks, it is often neither desirable nor even feasible to handle such a full proof—as in mathematics, concise lemmas or proof sketches are priceless.[25] Consequently, we support flexibility: Given two witnesses w_1 and w_2, we consider w_1 to be of higher quality than w_2 if a witness-based result validator can more quickly re-establish the verification result using w_1 than using w_2. A less detailed witness may still succeed in guiding the validator to the proof, but in turn may require more effort from the validator. Another difference to classic PCC is that we consider the witness as its own, separate, first-class object, and do not use the program to carry the proof, thereby following the best practice of separation of concerns, which leads to higher flexibility and maintainability.

7.7 Reusing Reachability Graphs

The intermediate results produced by model checkers during their state-space exploration are often materialized as an ARG [31], which consists of the abstract states found by the model checker and the program transitions between those states. The ARG is the basic data structure in tools like Blast and CPAchecker, and can be used as a source of invariants of the program [85], which in turn can be used for PCC, or for extreme model checking [86]. Extreme model checking checks if a previously computed ARG is also still a safety proof for a given, slightly modified, input program. Slab [70] is a certifying model checker that produces a proof certificate for the abstract model of a program in SMT-LIB format. While such a certificate can easily be checked using an SMT solver, mapping it back to the original program[26] to validate that it really certifies the correctness *of the original program* is non-trivial. As a result, even if checking the SMT-LIB certificate with an SMT solver produces the expected answer, a user still has no way to confirm that the certificate faithfully refers to the original program.

7.8 Search-carrying Code (SCC)

The concept of ***search-carrying code*** (**SCC**) [123] shares with verification witnesses the essential idea of reconstructing a verification result by guiding a validator through the state space of a program. For this purpose, SCC uses search scripts that guide a model checker along paths of

[25] The proof for the Schur-Number-Five problem is larger than 2 PB [88].

[26] In real-world scenarios, the original program is usually not given as a formal transition system with a well-defined one-to-one variable mapping to SMT-LIB, but must first be transformed by the verifier.

the ARG. Search scripts can be seen as a special instance of the generic concept of correctness witnesses where all invariants are omitted and the validator uses only the branching information from the witnesses as a suggestion to guide its state-space exploration, potentially saving time by simply confirming the suggested ARG rather than having to spend effort determining it itself from scratch. In comparison to search scripts, witnesses overcome the following three limitations (cf. Section 4.3 in [123]): (i) While SCC is bound to explicit-state model checking, the verification-witness exchange format is independent from the verification approach. (ii) The search scripts used by the existing implementation of SCC depend on a very specific transition-statement interpretation of **Java Pathfinder** (**JPF**), whereas verification witnesses allow a flexible mapping from program operations to the verifier-specific states and transitions that is even tolerant to code reduction, i.e., gaps in the witness that correspond to program code on which the producing verifier did not provide any information. (iii) Due to the reasons above, SCC is only supported by JPF, whereas the exchange format for verification witnesses is designed to work across different verifiers, even if they rely on different technologies, as shown by the widespread adoption of the format [12]. For practical impact, we have found these extensions to be essential.

7.9 Proof Programs and Configurable Certification

An important aspect of PCC is the goal that validation should be significantly faster than verification. In programs-from-proofs [96], correctness certificates are materialized as new programs that are behaviorally equivalent to the corresponding input program and are generated by a predicate analysis. Although they may be exponentially larger in terms of lines of code, these new programs can be verified by using a less expressive and more efficient data-flow analysis. Certificates for configurable program analysis [95, 96] consist of all reachable states of a program, which is comparable to a correctness witness where the reachable states are encoded as invariants at each program location. Various size-reducing techniques are then applied to reduce space consumption and I/O, and to speed up the validation. Because correctness witnesses do not require full proofs but can also contain partial proofs, a validator may choose to apply its own verification strategy to complement a partial proof or even perform the complete verification of the full verification task itself. As a consequence, correctness-witness-based validation of verification results does not necessarily exhibit a speedup. Nevertheless, in scenarios where the witnesses do represent complete proofs, similar techniques can be applied and speedups can be achieved. The size of correctness witnesses is generally not an issue in our case, because both implemented witness producers, CPAchecker and UAutomizer, restrict themselves to loop invariants and procedure post conditions instead of exporting invariants for every program location.

7.10 Partial Verification and Cooperative Verification

Software verifiers have three possible outcomes: they either (1) prove correctness, (2) detect a bug, or (3) fail. Correctness witnesses [23] and violation witnesses [25] address the first and second case, respectively. To complete the picture, ***conditional model checking*** (**CMC**) [32] addresses the third case. The idea of CMC is to provide reports of partial verification results in case full verification fails: An output condition describes the result of an incomplete verification attempt, i.e., which parts of the state space have already been verified successfully, and an input condition instructs a model checker to restrict the verification of a system, i.e., it describes which parts of the state space are left to be verified. To complete the verification, subsequent verification runs with a different approach can then use the output condition of the previous run as an input condition to simplify their task. Various concepts to represent the conditions, such as assumption automata [32] or execution reports [56], have been explored in existing implementations of CMC. Recently, the concept of reducer-based construction of conditional verifiers was introduced to

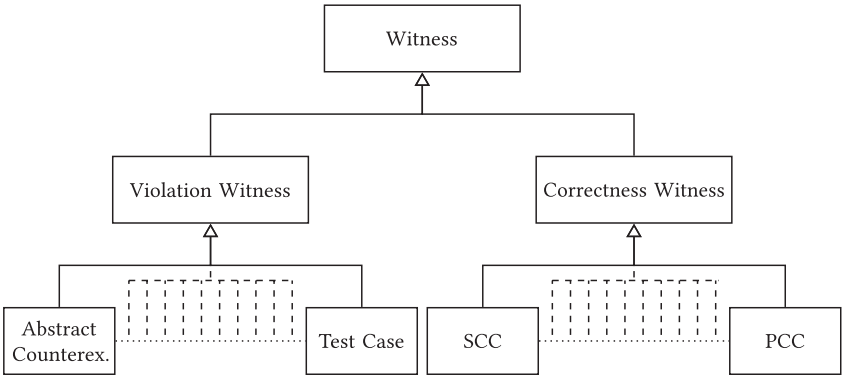


Fig. 24. Classification of different types of witnesses.

facilitate the adoption of CMC [37]: A *reducer* synthesizes, from a given input program and input condition, a new *residual* program that consists of only those parts of the original input program that are still to be verified, according to the input condition. Any off-the-shelf verifier can then be used for the conditional model checking of the residual program. As an alternative, verification witnesses could be used as a medium for CMC, by describing (a) paths (in violation witnesses) that hindered a complete verification and (b) invariants (in correctness witnesses) that were used to verify the part of the system that was successfully verified.

In cooperative verification [45], these techniques are leveraged to solve verification tasks by sharing information between different verification approaches and tools, not necessarily unidirectionally, but potentially even back and forth between components, and over multiple iterations [36].

7.11 Generalization

Verification witnesses subsume several of the previously known types of verification artifacts. We try to explain this using Figure 24. Firstly, we consider the two main types of witnesses disjoint, that is, a verification witness is either a violation witness or a correctness witness. This design choice is not obvious, because it is arguable why a violation witness should not contain invariants that help rule out considering infeasible error paths during the validation. Secondly, both witness types allow for a range of abstraction levels. A violation witness can be as abstract as an abstract counterexample from model checking (Section 7.3) on the one hand (abstract extreme: no restriction of data values, example: Figure 9(b)) and it can be as concrete as a test case (Section 7.4) on the other hand (concrete extreme: all data values concretely given, example: Figure 9(d)). But violation witnesses can have any level of abstraction in between the two extreme cases (intermediate: intervals for data values, example: Figure 9(c)). Similarly, a correctness witness can be as abstract as in a search-carrying code (Section 7.8) (only guiding the validator through the state space) and as concrete as in a proof-carrying code (Section 7.6) (providing all proof ingredients). There is a wide spectrum of possibilities in between, for example certificates (Sections 7.5 and 7.9).

8 CONCLUSION

Software verification, in general, is an undecidable problem. Therefore, effectiveness and efficiency have always been two main concerns of software verification, i.e., the goal was to make software verification solve more problems, and solve them quicker, and hence, there have been many breakthroughs in the past decades that made software verification efficient enough to be applicable on

an industrial scale. However, even though effectiveness and efficiency are certainly valid and important concerns, an oft-repeated argument against practical application of software verification is fear of the significant economic disadvantages caused by time wasted on the investigation of false alarms and of the potentially catastrophic consequences of misplaced trust in proofs that may turn out to be wrong. This fear is reinforced by a lack of usability of the results: A simple TRUE-or-FALSE answer to a verification problem is insufficient to understand and validate the result.

On the other hand, testing is an established method for software-quality assurance, because its results are concrete and tractable: An engineer constructs a test suite for a given coverage goal, executes the tests, and obtains precise and graspable results: (i) a quantitative coverage and (ii) a qualitative answer to the question which tests passed and which tests failed. This process requires considerable resources to be spent, but in return, concrete answers are provided, and, contrary to classic software verification, the interpretation of these results does not require an academic education.

If we compare the testing process to verification, we must acknowledge that in classic verification, an engineer also has to invest a significant amount of resources, as in testing, but in turn, gets back only an oversimplified answer TRUE or FALSE without any argument or explanation. The confidence in this answer is usually only derived from the reputation of the verification tool, because manually inspecting an error path for the verification answer FALSE to determine whether it represents an actual bug or a false alarm is a tedious task and a waste of expensive developer time. To make matters worse, most classic tools did not even bother to give an explanation why the verifier reports the program as correct when its answer to the verification problem is TRUE.

We aim at changing this situation and propose using tool-independent and machine-readable witnesses as a richer, more valuable form of verification result for both specification violations and correctness. In this article we presented a formalism to express both violation witnesses and correctness witnesses, while also outlining their necessary differences. We suggest a concrete format to represent such witnesses for verification results for tasks derived from C programs and present four different implementations of validators that support this format. We believe that producing witnesses should be easy, because in order to find a bug, a useful verifier should already be able to give the user a test case or a concrete error path, and any verifier designed for more than just falsification, i.e., hunting bugs, must also already derive some form of a proof of correctness. In practice, of course, there certainly are some engineering efforts required to construct a useful witness. Witness-based result validation, on the other hand, is more difficult to implement than witness construction: the validator must not only understand the assumptions and invariants in the witness, but also correctly assign them to the program states that they were intended for. The formalism presented in this article shows one possible approach for achieving this task, but if its direct implementation in a given verification framework is infeasible, the approach can be adapted, as exemplified by the different implementations that we showed.

We performed an extensive experimental study with thousands of verification runs on tasks from the largest public repository of verification problems (C programs). We implemented our validation approach in four result validators that have already been used for this purpose in the recent competitions on software verification, and have applied these validators to results produced by two verification tools that have achieved top scores in these competitions for years. The results obtained by our proof-of-concept implementations demonstrate that the proposed approach can work in practice. Since the advent of witnesses a few years ago, others have implemented support for witnesses in their tools. We hope that this process continues and that more developers find our ideas useful, thus adding the value of diversity to the concept: Although it may serve as a sanity check to apply a validator based on a certain framework to a witness produced by a verifier built on those same components, flaws in the reasoning may inadvertently be

covered up by a common defective component. Our solution is to instead establish a common exchange format supported by many verifiers, such that different result validators based on different technologies can be leveraged. In the meanwhile, there are eight published validators for C programs (CPAchecker [25], CPA-witness2test [26], Dartagnan [116], FShell-witness2test [26], MetaVal [44], NitWit [122], Symbiotic-Witch [4], and UAutomizer [25]), which are based on seven completely different technologies, and our current results on witness validation demonstrate that diversity is beneficial. In SV-COMP 2022, two validators for Java programs (Gwit [92] and Wit4Java [130]) were introduced. Establishing witnesses as an accepted standard in software verification will serve to open tools up to other uses besides plain verification and validation, such as quality measures for invariants or error paths, witness visualization, witness maintenance, databases for bugs and proofs, regression verification, and many more.

DECLARATIONS

Data-Availability Statement. All results, tools, and verification tasks that we used in our evaluation are available on the supplementary web page[27] and in a reproduction package at Zenodo [24]. In addition to the experimental results that we produced for this article, there are publicly available results for all verifiers and validators that participated in the competition on software verification (SV-COMP). A complete set of violation and correctness witnesses from the verifiers that participated in SV-COMP is available for 2019 [15] and for 2020 [17]. A statistical overview of the witnesses produced in SV-COMP 2019 and a description of the witness-store format that is used for those archives is also available [14]. Some of the results are shown in aggregated form in the competition reports, in Table 10 of the 2020 report [16], in Table 10 of the 2019 report [13], in Table 8 of the 2017 report [12], and in Table 7 of the 2016 report [11].

ACKNOWLEDGMENTS

We thank Martin Spiessl and the anonymous reviewers for their careful proof-reading and suggestions for improvement.

REFERENCES

[1] L. Alawneh and A. Hamou-Lhadj. 2011. MTF: A scalable exchange format for traces of high performance computing systems. In *Proceedings of the 19th International Conference on Program Comprehension*. IEEE, 181–184. https://doi.org/10.1109/ICPC.2011.15

[2] J. Alglave, A. F. Donaldson, D. Kröning, and M. Tautschnig. 2011. Making software verification tools really work. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis*. Springer, 28–42. https://doi.org/10.1007/978-3-642-24372-1_3

[3] C. Artho, K. Havelund, and S. Honiden. 2007. Visualization of concurrent program executions. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*. IEEE, 541–546. https://doi.org/10.1109/COMPSAC.2007.236

[4] P. Ayaziová, M. Chalupa, and J. Strejček. 2022. Symbiotic-Witch: A klee-based violation witness checker (competition contribution). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

[5] T. Ball, V. Levin, and S. K. Rajamani. 2011. A decade of software model checking with Slam. *Communication of the ACM* 54, 7 (2011), 68–76. https://doi.org/10.1145/1965724.1965743

[6] T. Ball and S. K. Rajamani. 2002. *SLIC: A Specification Language for Interface Checking (of C)*. Technical Report MSR-TR-2001-21. Microsoft Research. Retrieved from https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/.

[7] T. Ball and S. K. Rajamani. 2002. The Slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1–3. https://doi.org/10.1145/503272.503274

[27] https://www.sosy-lab.org/research/verification-witnesses-tosem/

[8] T. Balyo, M. J. H. Heule, and M. Järvisalo. 2017. SAT Competition 2016: Recent developments. In *Proceedings of the Proceedings of the AAAI Conference on Artificial Intelligence*. 31, 1 (2016), 5061–5063. https://doi.org/10.1609/aaai.v31i1.10641

[9] D. Beyer. 2012. Competition on software verification (SV-COMP). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 504–524. https://doi.org/10.1007/978-3-642-28756-5_38

[10] D. Beyer. 2015. Software verification and verifiable witnesses (report on SV-COMP 2015). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 401–416. https://doi.org/10.1007/978-3-662-46681-0_31

[11] D. Beyer. 2016. Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 887–904. https://doi.org/10.1007/978-3-662-49674-9_55

[12] D. Beyer. 2017. Software verification with validation of results (report on SV-COMP 2017). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20

[13] D. Beyer. 2019. Automatic verification of C and Java programs: SV-COMP 2019. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 133–155. https://doi.org/10.1007/978-3-030-17502-3_9

[14] D. Beyer. 2019. A data set of program invariants and error paths. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE, 111–115. https://doi.org/10.1109/MSR.2019.00026

[15] D. Beyer. 2019. *Verification Witnesses from SV-COMP 2019 Verification Tools*. Zenodo. https://doi.org/10.5281/zenodo.2559175

[16] D. Beyer. 2020. Advances in automatic software verification: SV-COMP 2020. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 347–367. https://doi.org/10.1007/978-3-030-45237-7_21

[17] D. Beyer. 2020. *Verification Witnesses from SV-COMP 2020 Verification Tools*. Zenodo. https://doi.org/10.5281/zenodo.3630188

[18] D. Beyer. 2021. Software verification: 10th comparative evaluation (SV-COMP 2021). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 401–422. https://doi.org/10.1007/978-3-030-72013-1_24

[19] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE, 326–335. https://doi.org/10.1109/ICSE.2004.1317455

[20] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. The Blast query language for software verification. In *Proceedings of the International Static Analysis Symposium*. Springer, 2–18. https://doi.org/10.1007/978-3-540-27864-1_2

[21] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. 2009. Software model checking via large-block encoding. In *Proceedings of the 2009 Formal Methods in Computer-Aided Design*. IEEE, 25–32. https://doi.org/10.1109/FMCAD.2009.5351147

[22] D. Beyer and M. Dangl. 2016. Verification-aided debugging: An interactive web-service for exploring error witnesses. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 502–509. https://doi.org/10.1007/978-3-319-41540-6_28

[23] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness witnesses: Exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

[24] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. 2020. *Reproduction Package for TOSEM Article "Verification Witnesses"*. Zenodo. https://doi.org/10.5281/zenodo.3731856

[25] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness validation and stepwise testification across software verifiers. In *Proceedings of the Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867

[26] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. 2018. Tests from witnesses: Execution-based validation of verification results. In *Proceedings of the International Conference on Tests and Proofs*. Springer, 3–23. https://doi.org/10.1007/978-3-319-92994-1_1

[27] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-induction with continuously-refined invariants. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 622–640. https://doi.org/10.1007/978-3-319-21690-4_42

[28] D. Beyer, M. Dangl, and P. Wendler. 2018. A unifying view on SMT-based software verification. *Journal of Automated Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[29] D. Beyer and K. Friedberger. 2020. Violation witnesses and result validation for multi-threaded programs. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*. Springer, 449–470. https://doi.org/10.1007/978-3-030-61362-4_26

[30] D. Beyer, S. Gulwani, and D. Schmidt. 2018. Combining model checking and data-flow analysis. In *Proeedings of the Handbook of Model Checking*. Springer, 493–540. https://doi.org/10.1007/978-3-319-10575-8_16

[31] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The software model checker Blast. *International Journal on Software Tools for Technology Transfer* 9, 5–6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[32] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional model checking: A technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 11 pages. https://doi.org/10.1145/2393596.2393664

[33] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. 2007. Path invariants. In *Proceedings of the Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 300–309. https://doi.org/10.1145/1250734.1250769

[34] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[35] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. 2013. Information reuse for multi-goal reachability analyses. In *Proceedings of the European Symposium on Programming*. Springer, 472–491. https://doi.org/10.1007/978-3-642-37036-6_26

[36] D. Beyer and M.-C. Jakobs. 2019. CoVeriTest: Cooperative verifier-based testing. In *Proceedings of the FASE*. Springer, 389–408. https://doi.org/10.1007/978-3-030-16722-6_23

[37] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. 2018. Reducer-based construction of conditional verifiers. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1182–1193. https://doi.org/10.1145/3180155.3180259

[38] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[39] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate abstraction with adjustable-block encoding. In *Proceedings of the Formal Methods in Computer Aided Design*. 189–197. Retrieved from https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf.

[40] D. Beyer and T. Lemberger. 2016. Symbolic execution with CEGAR. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*. Springer, 195–211. https://doi.org/10.1007/978-3-319-47166-2_14

[41] D. Beyer and T. Lemberger. 2019. TestCov: Robust test-suite execution and coverage measurement. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 1074–1077. https://doi.org/10.1109/ASE.2019.00105

[42] D. Beyer and S. Löwe. 2013. Explicit-state software model checking based on CEGAR and interpolation. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, 146–162. https://doi.org/10.1007/978-3-642-37057-1_11

[43] D. Beyer, S. Löwe, and P. Wendler. 2019. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21, 1 (2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y

[44] D. Beyer and M. Spiessl. 2020. MetaVal: Witness validation via verification. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 165–177. https://doi.org/10.1007/978-3-030-53291-8_10

[45] D. Beyer and H. Wehrheim. 2020. Verification artifacts in cooperative verification: Survey and unifying component framework. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*. Springer, 143–167. https://doi.org/10.1007/978-3-030-61362-4_8

[46] D. Beyer and P. Wendler. 2012. Algorithms for software model checking: Predicate abstraction vs. Impact. In *Proceedings of the 2012 Formal Methods in Computer-Aided Design*. 106–113. Retrieved from https://www.sosy-lab.org/research/pub/2012-FMCAD.Algorithms_for_Software_Model_Checking.pdf.

[47] D. Beyer and P. Wendler. 2013. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proceedings of the International SPIN Workshop on Model Checking of Software*. Springer, 1–17. https://doi.org/10.1007/978-3-642-39176-7_1

[48] P. Bielik, V. Raychev, and M. T. Vechev. 2017. Learning a static analyzer from data. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 233–253. https://doi.org/10.1007/978-3-319-63387-9_12

[49] 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

[50] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. 1999. Symbolic model checking without BDDs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[51] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. 2001. GraphML progress report. In *Proceedings of the Graph Drawing*. Springer, 501–512. https://doi.org/10.1007/3-540-45848-4_59

[52] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer. 2015. Facilitating reuse in multi-goal test-suite generation for software product lines. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, 84–99. https://doi.org/10.1007/978-3-662-46675-9_6

[53] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the CCS*. ACM, 322–335. https://doi.org/10.1145/1180405.1180445

[54] H. Cai, Z. Shao, and A. Vaynberg. 2007. Certified self-modifying code. *ACM SIGPLAN Notices* 42, 6 (2007), 66–77. https://doi.org/10.1145/1250734.1250743

[55] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. 2015. Moving fast with software verification. In *Proceedings of the NASA Formal Methods Symposium*. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[56] R. Castaño, V. A. Braberman, D. Garbervetsky, and S. Uchitel. 2017. Model checker execution reports. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 200–205. https://doi.org/10.1109/ASE.2017.8115633

[57] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. 2004. Modular verification of software components in C. *IEEE Transactions on Software Engineering* 30, 6 (2004), 388–402. https://doi.org/10.1109/TSE.2004.22

[58] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. 2016. The kind 2 model checker. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 510–517. https://doi.org/10.1007/978-3-319-41540-6_29

[59] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[60] E. M. Clarke, O. Grumberg, K. L. McMillan, and Xudong Zhao. 1995. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. ACM, 427–432. https://doi.org/10.1145/217474.217565

[61] E. M. Clarke and H. Veith. 2003. Counterexamples revisited: Principles, algorithms, applications. In *Proceedings of the Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*. Springer, 208–224. https://doi.org/10.1007/978-3-540-39910-0_9

[62] H. Cleve and A. Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, 342–351. https://doi.org/10.1145/1062455.1062522

[63] B. Cook. 2018. Formal reasoning about the security of amazon web services. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[64] P. Cousot and R. Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*. Dunod, 106–130. Retrieved from https://www.di.ens.fr/~cousot/COUSOTpapers/publications.www/CousotCousot-ISOP-76-Dunod-p106--130-1976.pdf.

[65] P. Cousot and R. Cousot. 1979. Systematic design of program-analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 269–282. https://doi.org/10.1145/567752.567778

[66] W. Craig. 1957. Linear reasoning. A new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic* 22, 3 (1957), 250–268. https://doi.org/10.2307/2963593

[67] C. Csallner and Y. Smaragdakis. 2005. Check "n" crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, 422–431. https://doi.org/10.1145/1062455.1062533

[68] D. Dams and K. S. Namjoshi. 2005. Orion: High-precision methods for static error analysis of C and C++ programs. In *Proceedings of the International Symposium on Formal Methods for Components and Objects*. Springer, 138–160. https://doi.org/10.1007/11804192_7

[69] A. F. Donaldson, L. Haller, D. Kröning, and P. Rümmer. 2011. Software verification using k-induction. In *Proceedings of the International Static Analysis Symposium*. Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[70] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. 2010. Slab: A certifying model checker for infinite-state concurrent systems. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 271–274. https://doi.org/10.1007/978-3-642-12002-2_22

[71] E. Ermis, M. Schäf, and T. Wies. 2012. Error invariants. In *Proceedings of the International Symposium on Formal Methods*. Springer, 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[72] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. 2011. Open trace format 2: The next generation of scalable trace formats and support libraries. In *Proceedings of the ParCo (APC 22)*. IOS, 481–490. https://doi.org/10.3233/978-1-61499-041-3-481

[73] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz. 2018. Executable counterexamples in software model checking. In *Proceedings of the Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 17–37. https://doi.org/10.1007/978-3-030-03592-1_2

[74] E. Ghassabani, A. Gacek, and M. W. Whalen. 2016. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 314–325. https://doi.org/10.1145/2950290.2950346

[75] J. Giesl, F. Mesnard, A. Rubio, R. Thiemann, and J. Waldmann. 2015. Termination competition (termCOMP 2015). In *Proceedings of the International Conference on Automated Deduction*. Springer, 105–108. https://doi.org/10.1007/978-3-319-21401-6_6

[76] P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[77] S. Graf and H. Saïdi. 1997. Construction of abstract state graphs with Pvs. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10

[78] A. Groce, S. Chaki, D. Kröning, and O. Strichman. 2006. Error explanation with distance metrics. *STTT* 8, 3 (2006), 229–247. https://doi.org/10.1007/s10009-005-0202-0

[79] A. Groce and W. Visser. 2003. What went wrong: Explaining counterexamples. In *Proceedings of the International SPIN Workshop on Model Checking of Software*. Springer, 121–135. https://doi.org/10.1007/3-540-44829-2_8

[80] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. 2006. SYNERGY: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 117–127. https://doi.org/10.1145/1181775.1181790

[81] E. L. Gunter and D. A. Peled. 1999. Path exploration tool. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 405–419. https://doi.org/10.1007/3-540-49059-0_28

[82] A. K. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. 2008. Proving non-termination. In *Proceedings of the annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 147–158. https://doi.org/10.1145/1328438.1328459

[83] 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 *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 394–398. https://doi.org/10.1007/978-3-662-54580-5_30

[84] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software model checking for people who love automata. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[85] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. 2002. Temporal-safety proofs for systems code. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 526–538. https://doi.org/10.1007/3-540-45657-0_45

[86] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. 2003. Extreme model checking. In *Proceedings of the Verification: Theory and Practice*. Springer, 332–358. https://doi.org/10.1007/978-3-540-39910-0_16

[87] M. J. H. Heule. 2016. The DRAT format and DRAT-trim checker. Tech. Rep. arXiv: 1610.06229. arXiv. https://doi.org/10.48550/arXiv.1610.06229

[88] M. J. H. Heule. 2018. Schur number five. *Proceedings of the AAAI Conference on Artificial Intelligence* 32, 1 (2018), 6598–6606. https://doi.org/10.1609/aaai.v32i1.12209

[89] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. 2010. How did you specify your test suite. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, 407–416. https://doi.org/10.1145/1858996.1859084

[90] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. 2003. Data flow testing as model checking. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 232–243. https://doi.org/10.1109/ICSE.2003.1201203

[91] H. S. Hong, I. Lee, and O. Sokolsky. 2001. *Automatic Test Generation From Statecharts Using Modle Checking*. Technical Report MS-CIS-01-07. University of Pennsylvania. 27 pages. Retrieved from https://repository.upenn.edu/cgi/viewcontent.cgi?article=1092&context=cis_reports.

[92] F. Howar and M. Mues. 2022. GWIT (competition contribution). In *Proceedings of the TACAS (2) (LNCS 13244)*. Springer.

[93] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. 2017. Safety verification of deep neural networks. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 3–29. https://doi.org/10.1007/978-3-319-63387-9_1

[94] A. Iliasov. 2011. Generation of certifiably correct programs from formal models. In *Proceedings of the 1st International Workshop on Software Certification*. IEEE, 43–48. https://doi.org/10.1109/WoSoCER.2011.14

[95] M.-C. Jakobs and H. Wehrheim. 2014. Certification for configurable program analysis. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. ACM, 30–39. https://doi.org/10.1145/2632362.2632372

[96] M.-C. Jakobs and H. Wehrheim. 2017. Programs from proofs: A framework for the safe execution of untrusted software. *ACM Transactions on Programming Languages and Systems* 39, 2 (2017), 7:1–7:56. https://doi.org/10.1145/3014427

[97] C. Jard and T. Jéron. 2005. TGV: Theory, principles, and algorithms. *STTT* 7, 4 (2005), 297–315. https://doi.org/10.1007/s10009-004-0153-x

[98] M. Jose and R. Majumdar. 2011. Bug-assist: Assisting fault localization in ANSI-C programs. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 504–509. https://doi.org/10.1007/978-3-642-22110-1_40

[99] T. Kahsai and C. Tinelli. 2011. PKind: A parallel k-induction based model checker. In *Proceedings of the Int. Workshop on Parallel and Distributed Methods in Verification*. EPTCS, 55–62. https://doi.org/10.4204/EPTCS.72.6

[100] A. V. Khoroshilov, V. S. Mutilin, A. K. Petrenko, and V. Zakharov. 2009. Establishing linux driver verification process. In *Proceedings of the International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14

[101] G. A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 194–206. https://doi.org/10.1145/512927.512945

[102] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. 2006. Introducing the open trace format (OTF). In *Proceedings of the International Conference on Computational Science*. Springer, 526–533. https://doi.org/10.1007/11758525_71

[103] D. Kröning and N. Sharygina. 2005. Formal verification of systemc by automatic hardware/software partitioning. In *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design*. IEEE, 101–110. https://doi.org/10.1109/MEMCOD.2005.1487900

[104] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. 2007. Efficient unit test case minimization. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 417–420. https://doi.org/10.1145/1321631.1321698

[105] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. 2012. Residual investigation: Predictive and precise bug detection. In *Proceedings of the ISSTA*. ACM, 298–308. https://doi.org/10.1145/2338965.2336789

[106] R. Majumdar and K. Sen. 2007. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE, 416–426. https://doi.org/10.1109/ICSE.2007.41

[107] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. 2011. Certifying algorithms. *Computer Science Review* 5, 2 (2011), 119–161. https://doi.org/10.1016/j.cosrev.2010.09.009

[108] K. L. McMillan. 2003. Interpolation and SAT-based model checking. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[109] K. L. McMillan. 2006. Lazy abstraction with interpolants. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 123–136. https://doi.org/10.1007/11817963_14

[110] P. Müller and J. N. Ruskiewicz. 2011. Using debuggers to understand failed verification attempts. In *Proceedings of the International Symposium on Formal Methods*. Springer, 73–87. https://doi.org/10.1007/978-3-642-21437-0_8

[111] K. S. Namjoshi. 2001. Certifying model checkers. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 2–13. https://doi.org/10.1007/3-540-44585-4_2

[112] G. C. Necula. 1997. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 106–119. https://doi.org/10.1145/263699.263712

[113] F. Nielson, H. R. Nielson, and C. Hankin. 1999. *Principles of Program Analysis*. Springer. https://doi.org/10.1007/978-3-662-03811-6

[114] Evgeny Novikov and Ilja S. Zakharov. 2017. Towards automated static verification of GNU C programs. In *Proceedings of the International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 402–416. https://doi.org/10.1007/978-3-319-74313-4_30

[115] OASIS. 2019. Static Analysis Results Interchange Format (SARIF) Version 2.0. Retrieved 23 July 2022 from https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.html.

[116] H. Ponce-De-Leon, T. Haas, and R. Meyer. 2022. Dartagnan: SMT-based violation witness validation (competition contribution). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

[117] H. O. Rocha, R. S. Barreto, L. C. Cordeiro, and A. Dias Neto. 2012. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *Proceedings of the International Conference on Integrated Formal Methods*. Springer, 128–142. https://doi.org/10.1007/978-3-642-30729-4_10

[118] F. B. Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

[119] K. Sen, D. Marinov, and G. Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the ACM SIGSOFT Software Engineering Notes*. ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[120] O. Šerý. 2009. Enhanced property specification and verification in BLAST. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, 456–469. https://doi.org/10.1007/978-3-642-00593-0_32

[121] C. Sternagel and R. Thiemann. 2014. The certification problem format. In *Proceedings of the UITP (EPTCS 167)*. EPTCS, 61–72. https://doi.org/10.4204/EPTCS.167.8

[122] J. Švejda, P. Berger, and J.-P. Katoen. 2020. Interpretation-based violation witness validation for C: NITWIT. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 40–57. https://doi.org/10.1007/978-3-030-45190-5_3

[123] A. Taleghani and J. M. Atlee. 2010. Search-carrying code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, 367–376. https://doi.org/10.1145/1858996.1859079

[124] A. Turing. 1949. Checking a large routine. In *Proceedings of the Report on a Conference on High Speed Automatic Calculating Machines*. Cambridge Univ. Math. Lab., 67–69. Retrieved from http://dl.acm.org/citation.cfm?id=94938.94952.

[125] W. Visser, C. S. Păsăreanu, and S. Khurshid. 2004. Test-input generation with Java PATHFINDER. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 97–107. https://doi.org/10.1145/1007512.1007526

[126] M. Wagner, A. Knüpfer, and W. E. Nagel. 2016. OTFX: An in-memory event tracing extension to the open trace format 2. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 3–17. https://doi.org/10.1007/978-3-319-49956-7_1

[127] T. Wahl. 2013. The k-Induction Principle. Retrieved 23 July 2022 from http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf.

[128] N. Wetzler, M. J. H. Heule, and Warren A. Hunt Jr. 2014. DRAT-TRIM: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Springer, 422–429. https://doi.org/10.1007/978-3-319-09284-3_31

[129] M. Whalen, J. Schumann, and B. Fischer. 2002. Synthesizing certified code. In *Proceedings of the International Symposium of Formal Methods Europe*. Springer, 431–450. https://doi.org/10.1007/3-540-45614-7_25

[130] T. Wu, P. Schrammel, and L. Cordeiro. 2022. WIT4JAVA: A violation-witness validator for Java Verifiers (Competition Contribution). In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

[131] A. Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM SIGSOFT Software Engineering Notes*. ACM, 1–10. https://doi.org/10.1145/587051.587053

Received 15 August 2019; revised 17 July 2021; accepted 26 July 2021

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.,

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

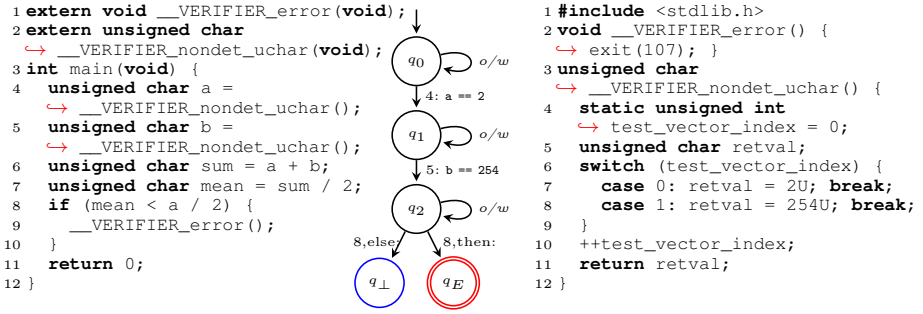
in the past, static analyzers reported too many bugs that were not observable in a concrete program execution, and thus, developers have lost confidence in bug reports [20]; (b) there is a lack of appropriate interfacing, i.e., it is difficult for developers to leverage advantages of the verification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



(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

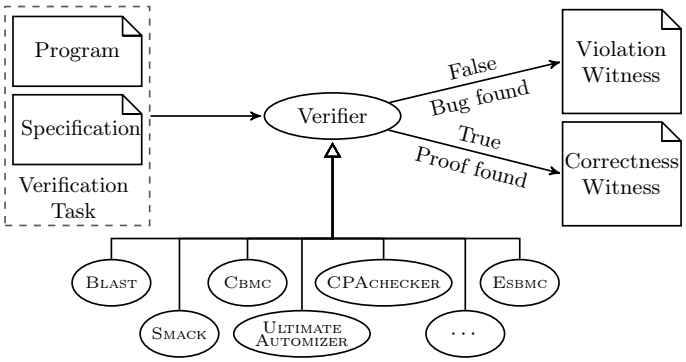
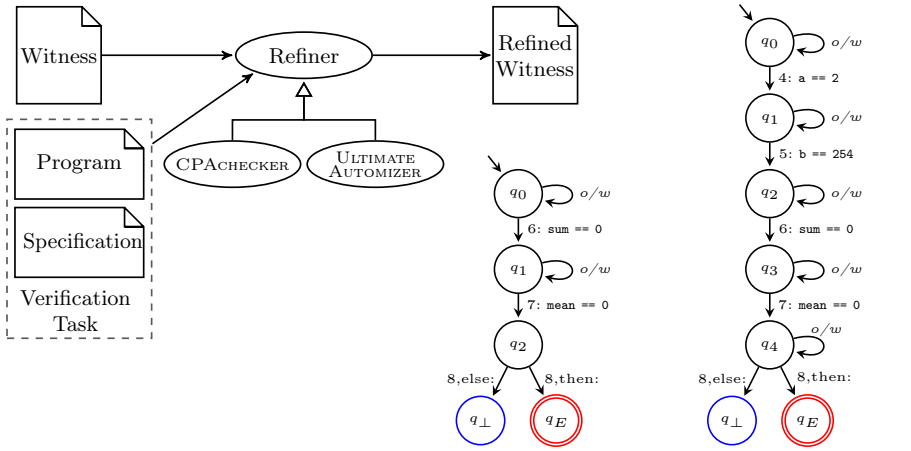


Fig. 2. Software verifiers produce witnesses



(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.

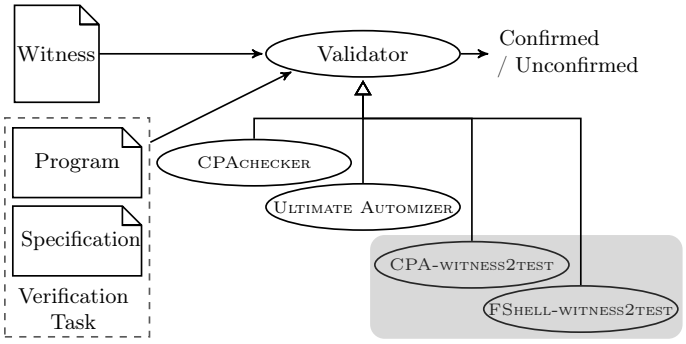


Fig. 4. Violation-witness validation

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,

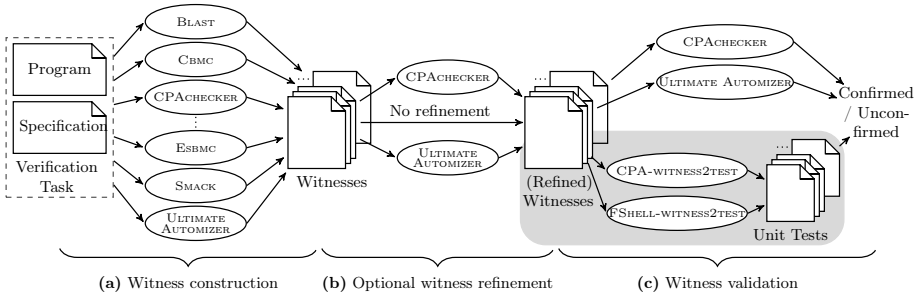


Fig. 5. Software verification with witnesses: construction, (optional) refinement, and validation work flow

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.

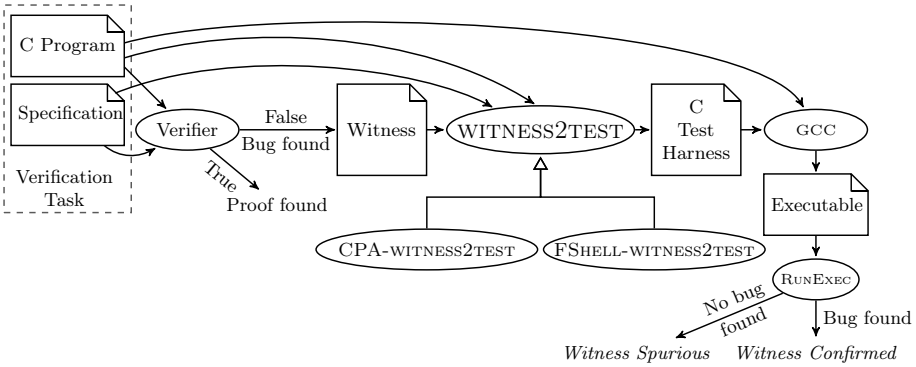


Fig. 6. Flow of execution-based result validation

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 CPACHECKER 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.

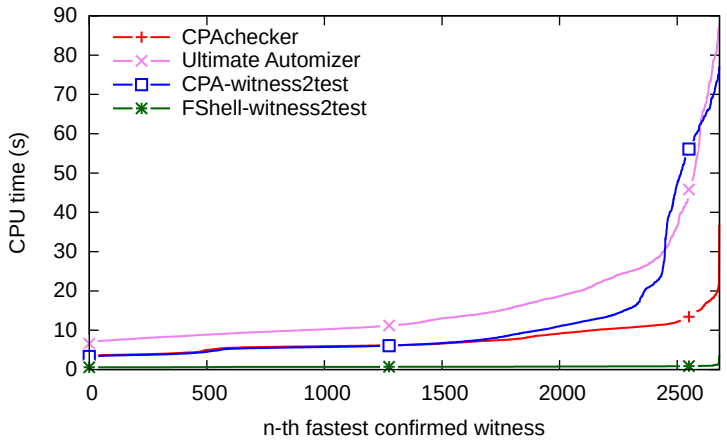


Fig. 7. Quantile plot for CPU time consumed for validating witnesses accepted by all validators

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.

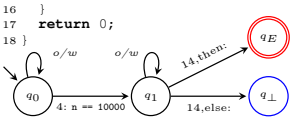
Claim 3: Test Generation. The last four columns of Table 1 relate the number of witnesses that we processed to the number of produced tests for which failing executions are realizable. With 'produced tests' we refer to the tests that were produced by any of the dynamic validators and for which the test 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.

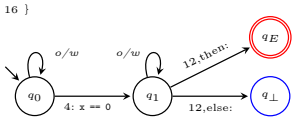
Figure 8a shows an example program intended to compare the iterative sum of ascending values with the result of the Gauss sum formula, and a witness for a bug in the program. The bug is located in lines 10 to 12 and causes an error for inputs larger than or equal to 10 000. The depicted witness for this 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

```
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 }
```



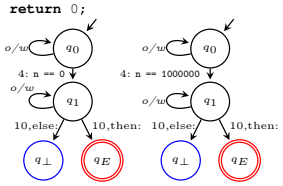
(a) "gauss" code, witness

```
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

```
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)

Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses

Dirk Beyer and Matthias Dangl

University of Passau, Passau, Germany

Abstract. Traditionally, a verification task is considered solved as soon as a property violation or a correctness proof is found. In practice, this is where the actual work starts: Is it just a false alarm? Is the error reproducible? Can the error report later be re-used for bug fixing or regression testing? The advent of *exchangeable witnesses* is a paradigm shift in verification, from simple answers *true* and *false* towards qualitatively more valuable information about the reason for the property violation. This paper explains a convenient web-based toolchain that can be used to answer the above questions. We consider as example application the verification of C programs. Our first component collects witnesses and stores them for later re-use; for example, if the bug is fixed, the witness can be tried once again and should now be rejected, or, if the bug was not scheduled for fixing, the database can later provide the witnesses in case an engineer wants to start fixing the bug. Our second component is a web service that takes as input a witness for the property violation and (re-)validates it, i.e., it re-plays the witness on the system in order to re-explore the state-space in question. The third component is a web service that continues from the second step by offering an interactive visualization that interconnects the error path, the system's sources, the values on the path (test vectors), and the reachability graph. We evaluated the feasibility of our approach on a large benchmark of verification tasks.

1 Introduction

The answer of a verification tool to a given verification task (consisting of a specification and a system) is either that the system satisfies the specification or that the system violates the specification (or the answer 'unknown' is returned) [9]. If a violation of the specification is detected, an error path through the system is reported that exhibits the problem, such that the user can understand the problem and fix the bug: counterexamples to verification have been described as invaluable to debugging complex systems and have been a common feature of model checkers for several decades [7]. In particular, the successful technique of counterexample-guided abstraction refinement (CEGAR) [8] is based on analyzing error paths through the system.

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part II, LNCS 9780, pp. 502–509, 2016.
DOI: 10.1007/978-3-319-41540-6_28

In the past few years, there was a strong focus in the community on using common exchange formats and reproducing errors described by previously computed counterexamples. ESBMC was extended to reproduce errors via instantiated code [11], and CPACHECKER was used to re-check previously computed error paths by interpreting them as automata that control the state-space search [6]. While these internal approaches to witness validation can reduce the amount of false alarms reported by a tool, they establish no additional trust in a report produced and validated by an untrusted verifier. The advantages of considering error witnesses as a valuable verification artifact were explained and supported by two completely different implementations of witness validators [4], namely CPACHECKER and AUTOMIZER. Also, competitions in the community required exchangeable witnesses: the competition on termination uses a certification-problem format (CPF)[1] and the competition on software verification uses a machine-readable, exchangeable format for error witnesses[2]. Our toolchain is based on the common exchange format that was used in SV-COMP [2,4], which allows specifying counterexample traces using control-flow paths and data values. Previous efforts towards helping users understand the counterexamples have lead to interactive trace visualizations [1,5,10], but the user was locked-in to a certain toolchain. The introduction of machine-readable error witnesses has opened up new possibilities for collecting, accumulating, and validating counterexample traces from different verifiers [4]. A wide range of software verifiers already supports a common exchange format, as shown by the competition on software verification[3], which has adopted error-witness validation already two years ago.

Error witnesses support traditional debugging very well: the test values that a witness might contain can direct a classic debugger through the system to the problematic part of the implementation or model. But the exchangeable witnesses support even a more abstract form of debugging, based on a graphical visualization of error paths and reachability graphs.

Figure 1 gives an overview over the components involved in our toolchain. There are three subsystems that the user interacts with: (1) We developed a *witness store* for persistently keeping error witnesses that different verification tools have produced. The database enables the user to select and retrieve specific witnesses for a given set of verification tasks. One possible use case is to fetch all witnesses that document a bug in a specific C program, to help the developer better understand the issue. (2) We offer an *online witness validator* with a convenient web-service API that enables validation without the need to install software. A bug report that a verifier returns can potentially be a false alarm, so it is convenient for the user to first automatically cross-examine the report, before manual effort is invested (and perhaps wasted). To validate an error witness, the user can send the validation task, which consists of the source-code file, the property, and a corresponding error witness (potentially obtained from the

[1] http://cl-informatik.uibk.ac.at/software/cpf
[2] http://sv-comp.sosy-lab.org/2016/witnesses/
[3] For example, see the list of systems in SV-COMP 2016 http://sv-comp.sosy-lab.org/2016/systems.php.

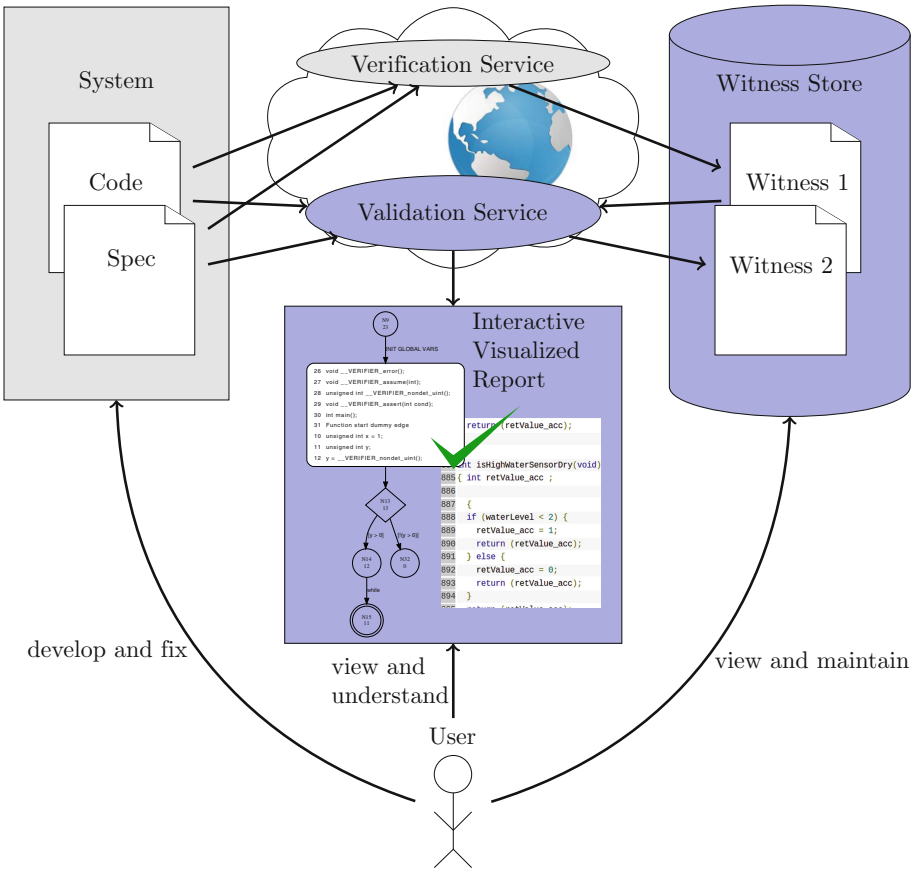


Fig. 1. System overview, blue parts are discussed in this paper (Color figure online)

witness database), to the validation service. The service then validates the error witness. If the witness is rejected, the user is advised to prioritize other tasks, because the specific error path that the witness describes has been declared as infeasible. If, instead, the witness is validated, the validation service feeds all information gained about the bug into the third component, the interactive report. (3) Successful witness validations produce a detailed and interactive web-based bug report. The report contains a debugger-like feature for stepping through the error path, while providing several context-sensitive representations of the buggy program. The report also encompasses all information required to reproduce the validation externally.

Application Example. Our application example is the verification of system programs written in the language C. While the concepts of our toolchain can be applied to other programming languages, we restrict our tools to C. The web service that we describe is available on the internet, and our primary target is to

support open-source projects. Organizations that develop proprietary software can still benefit from our system, because it is easily installed on a local web server that is restricted to the organization's intranet.

Data to Experiment. As part of our evaluation, we ran several verification tools that participated in the competition on software verification (because those tools are known to generate useful witnesses) and fed the witnesses into our database. For the reader to assess our toolchain, we have compiled an archive with witnesses, validation results and error-path visualizations for offline use. The archive is available as supplement, and the validation results and visualization results can be reproduced via our live web service or offline using the CPAchecker-based witness validator[4] The archive contains reports for a total of 1 382 witnesses for 26 verification tasks that contain a bug. The average number of witnesses that we collected is 53 witnesses per verification task, the program with the fewest has 4, the program with the most has 114 witnesses in our database.

2 Collection of Error-Paths in a Witness Store

We consider witnesses as a prime-value verification artifact, because they can make it (a) efficient to re-run a partial verification to explore the bug again and (b) easy to use different verification tools for validation.

Permanently storing witnesses opens many new practical applications to let verification technology have a larger impact on system development. Our witness store provides a means to take advantage of the various beneficial properties of machine-readable witnesses in a common exchange format:

- Witness Validation: Imprecise verifiers may sometimes produce false alarms and thus waste valuable developer time. With witness validation, users no longer need to trust the answer False. Instead, they can concentrate on paying attention to witnesses that are confirmed by an automatic witness validator. Each validation run that confirms a witness can increase the user's confidence in the bug report.
- Witness Inspection: Witness validators with complementing strategies can be applied to a witness, each leveraging its strengths to add diagnostic information that the others may be incapable to derive. Therefore, witness validation can be understood as a chain of ever refining details for identifying, understanding, and fixing the bug.
- Bug Reports: In bug reports, attached witnesses can be used to provide a precise description of the erroneous behavior, including test-vector values.
- Re-Verification: Working with error witnesses is cheap in terms of resources, because the verification result can often be re-established with reduced effort.

[4] The URL to our supplementary web page, which includes the live web service, the archive for offline use, and a virtual machine set up for validating the witnesses and reproducing the results using CPAchecker 1.6, is: https://www.sosy-lab.org/~dbeyer/witness-based-debugging/.

This is not only beneficial for validating a given witness, but also when checking for regressions: If the witness is still valid for a changed version of the system, the bug has been reintroduced or was not yet fixed [6].

3 Convenient Witness Validation

A witness validator is a verifier that analyzes the synchronized product of the system with the witness automaton, where transitions are synchronized using system operations and transition annotations. This means that the witness automaton observes the system paths that the verifier wants to explore: if the operation on the system path does not match the transition of the witness automaton, then the verifier is forbidden to explore that path further; if the operation on the path matches, then the witness automaton and the system proceed to the next state, possibly restricting the system's state such that the assumptions given in the data annotation are satisfied. Implementations of witness validators are available, see for example CPAchecker and Automizer [4]. Our validation service uses the CPAchecker witness validator as back-end. CPAchecker supports and combines many different verification strategies, for example value analysis, predicate abstraction, CEGAR, bounded model checking, k-induction, and concrete memory graphs. The specific configuration that is effectively used to validate the witnesses via our web service is bit-accurate and combines value analysis and predicate abstraction. Our web service does not yet support arrays, concurrency, and termination analysis.

Conceptually, an *error-witness automaton* is a protocol automaton, and an *error-witness analysis* is a protocol analysis for an error-witness automaton [4], which runs as a component of a composite program analysis. Unlike observer automata [3], which can be used to represent the specification the analyzed program is verified with, error-witness automata not only observe the state-space exploration of the program analysis, but also *restrict* it to those successor states that lead the exploration toward a specification violation, whereas an observer automaton follows all abstract successor states. Therefore, the program analysis is *guided* by the error-witness automaton to explore the state space that violates the specification.

The process of determining if it is possible to independently re-establish a verification result, given the program, specification, result, and witness, is called *witness validation*. One way of implementing error-witness validation is by constructing a composite program analysis that has both a witness analysis and a specification analysis as components, which simultaneously restrict and observe the state-space exploration: the specification analysis checks if an analyzed path actually violates the specification, and the search of the composite program analysis is restricted by the witness validation such that only paths that the error-witness automaton can match are explored. For example, the analysis stops exploring a path, if, during the analysis of that path, the witness automaton takes a transition to a sink state. An error witness is confirmed by the witness validator if both, the witness automaton and the specification automaton, take a transition to their respective (accepting) error state [4].

4 Visualizing and Interactively Exploring Error-Paths

Figure 2 shows a screenshot of an interactive counterexample report. The screen is divided into two columns: The left column provides detailed information that is specific to the error path, namely the source code on the path to the property violation, and, like in a debugger, the program locations are decorated with test values that were computed by the witness validator. The right column embeds the specific information from the left column into the general context of the system and the analysis. It contains control-flow automata (CFA) for each of the functions, the abstract reachability graph (ARG) of the verification, full source code of the verification task, the verification log, statistics, and configuration parameters of the validation run. In all CFA and the ARG, the states on the path to the property violation are marked in red. Double clicking on a control-flow state that precedes a function call displays the CFA of the called function. Both columns, however, are not only useful in isolation: clicking on a line of code in the left column while viewing the ARG or CFA will navigate to the state corresponding to the clicked line of source code.

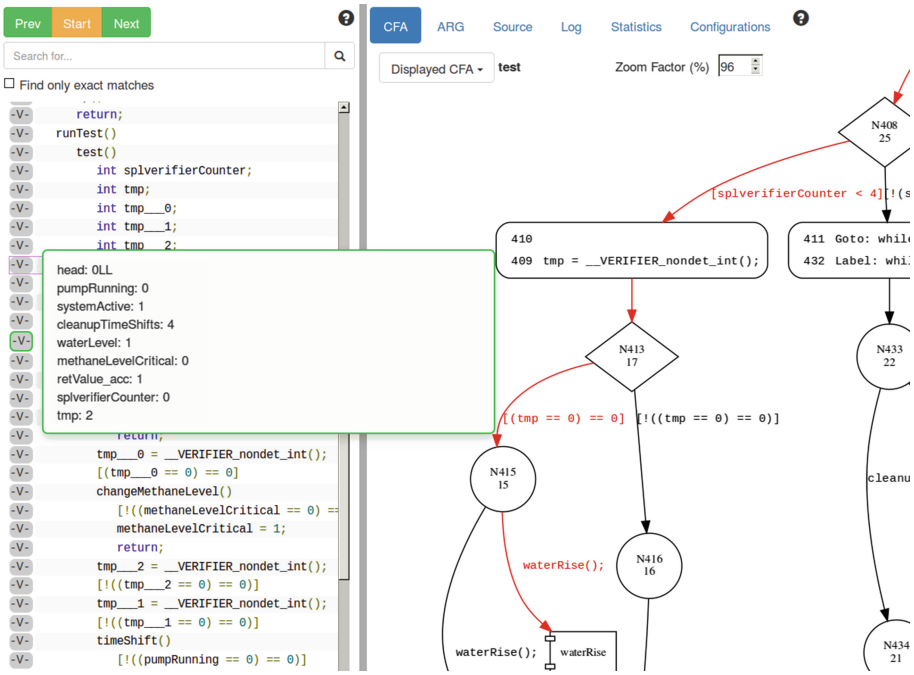


Fig. 2. Typical view of the error-path visualizer: program source code with violating test vector (left, green) and CFA with violating path (right, red); left view top shows the menu for debugger-like step-through, right view top shows the display options: CFA, ARG, Source, Log, Statistics, Configurations (Color figure online)

The visualization is built upon the JavaScript framework AngluarJS and the jQuery and Bootstrap web-development libraries. The layout of the graphs is computed using GraphViz and exchanged in SVG format. The complete data for one such error-path visualization takes on average 120 kB of memory.

5 Conclusion

Over the past decades, the algorithmic abilities of verification tools were considerably increased, but in practice, verification technology is still not as popular as testing. Why? Because it is inconvenient to use. Our work contributes to closing this gap, by considering not only the true/false answers as value, but actively using other results of the verification process, most prominently the error witnesses. We have presented a toolchain that supports engineers in understanding the error reports of verification systems. First, we archive verification witnesses permanently in a database. Second, we provide a convenient web service for witness validation, i.e., a verification task together with a witness can be given as input, and the results are presented via the web API (for manual inspection or automatic retrieval). Third, we explain an error-path visualization that supports an interactive investigation of the source code, the control-flow graph, the reachability graph, and test values. We believe that the proposed method is a step towards a more convenient usage of verification results.

References

1. Aljazzar, H., Leue, S.: Debugging of dependability models using interactive visualization of counterexamples. In: Rubino, G. (ed.) Proc. QUEST 2008, pp. 189–198. IEEE (2008)
2. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In: Chechik, M., Raskin, J.-F. (eds.) Proc. TACAS 2016. LNCS, vol. 9636, pp. 887–904. Springer, Heidelberg (2016)
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast query language for software verification. In: Giacobazzi, R. (ed.) Proc. SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
4. 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.) Proc. FSE 2015, pp. 721–733. ACM (2015)
5. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
6. Beyer, D., Wendler, P.: Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In: Bartocci, E., Ramakrishnan, C.R. (eds.) Proc. SPIN 2013. LNCS, vol. 7976, pp. 1–17. Springer, Heidelberg (2013)
7. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: Algorithmic verification and debugging. Commun. ACM **52**(11), 74–84 (2009)
8. 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)

9. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
10. Groce, A., Kröning, D., Lerda, F.: Understanding counterexamples with `explain`. In: Alur, R., Peled, D.A. (eds.) Proc. CAV 2004. LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004)
11. 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.) Proc. IFM 2012. LNCS, vol. 7321, pp. 128–142. Springer, Heidelberg (2012)