Leveraging Invariant Information towards Incremental Software Model Checking

Dissertation

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

eingereicht von

Martin Spiessl

12.10.2023

1. Gutachter:	Prof. Dr. Dirk Beyer
2. Gutachter:	Prof. Dr. Joost-Pieter Katoen
3. Gutachter:	Prof. doc. RNDr. Jan Strejček
Tag der mündlichen Prüfung:	22.05.2024

Eidesstattliche Versicherung

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

München, **12.10.2023**	**Martin Spiessl**
Ort, Datum	Unterschrift

Abstract

Automatic Software Model Checking has become more and more powerful over the recent years. While the tools have improved, the means of exchange of information as well as the kind of information that is exchanged, are lacking behind. This is a limiting factor for progress in the field, as a lot of potential lies in the exchange and careful dissemination of computed information.

Currently, information about the proof or counterexample can be extracted in the form of witness automata that refer to the program's control-flow automaton (CFA). These witnesses can then be checked by a validator, which is an important step that increases the trust in the verification result. However, this format suffers from imprecision in its semantics, since the CFA of a program is usually not defined by the language standard. Furthermore, in its current form, the format is syntactically limited to invariants that can be written as expressions in the program's language. This is in strong contrast to the deductive verification community, where proofs usually require more elaborate syntactical features. Lastly, the way how this information is (re-)used is very limited. Until recently, the only use case was the validation of verification results. In case the validation fails, it is often unclear how to gain insights from that.

We improve upon the state of the art in the following ways: Regarding the exchange format, we identify the areas in which the established format is imprecise and evaluate whether we can improve upon this. As solution, we propose an alternative format whose semantics is well-defined. Using this format, we showcase how extensions can be gradually incorporated by automatic software verifiers.

With respect to exchanging information between tools, we first show a way how information from the witnesses can be encoded back into a verification problem. As an extension of this, we demonstrate how this approach can be used to construct a deductive verifier from existing, automatic verifiers, using the witnesses as an established interface. This way, verification results can be reused in an incremental fashion more easily or even complemented via user interaction.

Zusammenfassung

Automatisches Software-Model-Checking ist in den letzten Jahren immer leistungsfähiger geworden. Während sich die Tools verbessert haben, haben sich die Austauschformate für Informationen sowie die Art der ausgetauschten Informationen nicht in selbem Maße weiterentwickelt. Dies ist ein limitierender Faktor für Fortschritte in diesem Bereich, da gerade jüngste Ansätze zeigen, dass viel Potenzial im Austausch und der Wiederverwendung der berechneten Informationen liegt.

Derzeit können Informationen über den Beweise oder Gegenbeispiele in Form von Automaten extrahiert werden, die sich auf den Kontrollflussautomaten (CFA) des Programms beziehen. Diese Zeugen können dann von einem sogenannten Validator überprüft werden, was ein wichtiger Schritt ist, um das Vertrauen in das Verifizierungsergebnis zu erhöhen. Dieses Format leidet jedoch unter einer unvollständigen Spezifikation seiner Semantik, da der CFA eines Programms in der Regel nicht durch den Sprachstandard definiert ist. Des Weiteren ist das Format in seiner derzeitigen Form syntaktisch auf Invarianten beschränkt, die als Ausdrücke in der Sprache des Programms geschrieben werden können. Das steht im starken Kontrast zur Gemeinschaft rund um die deduktiven Verifikation, wo Beweise normalerweise ausführlichere syntaktische Merkmale erfordern. Schließlich ist die Art und Weise, wie diese Informationen wiederverwendet werden, sehr eingeschränkt. Bis vor kurzem war der einzige Anwendungsfall die Validierung von Verifizierungsergebnissen. Falls die Validierung fehlschlägt, ist es oft unklar, wie man daraus Erkenntnisse gewinnen kann.

Wir verbessern den Stand der Technik in folgender Weise: Bezüglich des Austauschformats identifizieren wir die Bereiche, in denen das etablierte Format ungenau ist und untersuchen, in welcher Weise diese Ungenauigkeit behoben werden kann. Als Lösung schlagen wir ein alternatives Format vor, dessen Semantik klar definiert ist. Mit diesem Format zeigen wir, wie Erweiterungen schrittweise von automatischen Programmverifizierern integriert werden können.

In Bezug auf den Informationsaustausch zwischen den Verifikations-Tools zeigen wir zuerst, wie Informationen aus den Witnesses in ein Verifizierungsproblem zurück kodiert werden können. Als Erweiterung davon zeigen wir, wie dieser Ansatz verwendet werden kann, um einen deduktiven Verifier aus bestehenden automatischen Verifiern zu erstellen, wobei die Witnesses als etablierte Schnittstelle dienen. Auf diese Weise können Verifizierungsergebnisse leichter schrittweise wiederverwendet werden oder sogar durch Benutzerinteraktion ergänzt werden.

Acknowledgement

All this would not have been possible without the inspiration and heart-felt support of numerous people which I hold dear. So I want to do the least here, which is to express my deep gratitude for the support I received.

First and foremost, I want to thank my family and friends for their love, guidance, support, and their trust in me. My mother, who raised me to be able to pursue my dreams and who loves me, no matter what. My father, who sparked the interest for technology inside me, and who probably understands me the best. Charly, who has the heart in the right spot, accepts me as I am, and whom I can always rely on when I have a problem. And finally my sister Katrin, my brother Chris, and my nephew Maximilian, who always make me smile with their welcoming nature.

I also want to thank my academic family, led by my supervisor and mentor, Dirk Beyer, who helped me improve myself both professionally and personally. He was always there for when I encountered a roadblock, and served as a role model for me. I feel like I had the opportunity to learn from one of the best, which I do not take for granted. I will always look back to the time at his Chair for Software and Computational Systems (SoSy-Lab) with fond memories. Memories which are also accompanied by the fine colleagues and students whom I had the honor to work with, in one way or another. I want to thank you all for teaching me the true value of collaboration and companionship.

Of course there are also several professors I want to thank: Joost-Pieter Katoon and Jan Strejcek, for serving as examiners for this thesis. I am grateful to have met Martin Wirsing and Rolf Hennicker, without them I would most-likely never have ended up in the field of formal verification.

Almost from the start of my doctoral studies, I had the chance to join the ConVeY research training group (DFG GRK 2428), where I could connect to peers from different research groups in Munich. I am grateful for this opportunity, and want to thank all ConVeY members for their inspiration, as well as DFG for making this possible. A special thanks goes to Prof. Helmut Seidl, who did an immaculate job in leading the research training course, to Michael, the representative of the doctoral researchers, for always having an open ear, and cheering everyone up with his positive attitude, and of course to Niloofar and Mahathi, for their friendship and the countless memories that we formed together.

Speaking of friendship, there is a saying that you are the average of your five closest friends. Ben, Caro, Dani, Daniel, and Gerli – thank you for shaping this journey in more ways than you know.

Contents

1. Introduction

"Computers are incredibly fast, accurate, and stupid.
Human beings are incredibly slow, inaccurate, and brilliant.
Together they are powerful beyond imagination."

– often attributed to Albert Einstein

The quote at the beginning of this chapter offers us a way to explore the kind of dilemma we want to address in this work. While the quote is often attributed to Albert Einstein, these words are actually not his. Instead, there are several versions of this quote that evolved over the years and at some point someone either misremembered the author or deliberately put Einstein as the author to improve their standpoint.

People use attribution as a way to increase the power of a statement. A clever-sounding quote is improved by adding a well-known and respected name to it. However, the actual impact a quote has should only rely on the actual information it brings across. This perfectly carries over to the contrast the quote in question puts between humans and computers. Just because a computer is incredibly accurate, how can a human trust its computations to be correct, as it is also very stupid and only executes the instructions that the human provided? In a way, the computer cannot be blamed if the human operator writes faulty code since humans are often inaccurate.

A system in which one blindly trusts either side of this equation will inevitably lead to errors. Nowadays, where computers are used in almost every aspect of our daily lives, from entertainment and payment systems to healthcare and avionics, it is more important than ever to ensure that both sides *cooperate* to ensure the solutions are standing on a sound foundation.

The branch of computer science that investigates ways to ensure that a computer program works as intended is called *formal methods*.

1.1 Motivation

In the recent decades, two different approaches emerged in formal methods. On the one hand, people try to manually – or with the help of a proof assistant – craft proofs of correctness for certain programs, especially in safety-critical areas [53, 55]. We shall refer to this branch as *deductive verification* [49], though sometimes the process will also be described as interactive verification, due to the nature of the process that revolves around the interaction between the human and the proof assistant. On the other hand, computer systems have been developed that can check other computer systems for correctness in a fully automated manner, which we shall refer to under the name *automatic verification*.

For ensuring soundness of the computed results, both directions employ different means. In deductive verification, usually one relies on a small core for the proof system whose soundness can either be assumed or (at least partially) formally proven. For example, the automated theorem prover Isabelle has a small logic core based on the LCF approach [67]. For some proof assistants like Coq, there are even successful efforts to prove at least parts of the kernel correct [63]. Another approach is to check the proof objects themselves, removing the requirement to trust the proof assistant all together. For Isabelle, such a proof checker has recently been demonstrated [58].

For automatic verifiers, the picture is a little bit different. While exporting and checking full proofs in the form of reachability graphs has been shown to be possible, this is highly tool-dependent and not suitable for an exchange format for validation by another tool. Instead, the community opted to output partial proof information in the form of so-called verification witnesses, which take the form of automata and are exported by verifiers in the GraphML format [14]. One of the problems here is that there is no minimum requirement on which information needs to be present, so in the worst case such a witness will not provide any more information than claiming that the program adheres to the specification.

In this thesis, we also attempt to bridge the gap between these two schools of thought and improve cooperation by concentrating on the crucial bits of information, namely, the invariants.

One important aspect about validating verification results has not been mentioned yet. It is crucial in software verification to accurately capture the semantics of the underlying programming language, which is a hard and tedious process. Even if results are formally verified and independently validated, there is no guarantee that there is not a bug that stems from edge cases in the semantics of the programming language.

Ideally, validation is well-defined and relatively simple, such that we can concentrate and argue about soundness issues without having to deal with intricacies of the complicated validation strategy.

1.2 Objectives

The previous section shed some light on our motivation, but the objectives might hide between the lines there, and of course, we cannot address everything. For this thesis, we therefore focus on the following three objectives:

1. We want to leverage the current exchange formats such that we can come up with novel ways to combine existing tools, leading to inter-tool cooperation (cf. Sect. 2.1)

2. We want to know how the information that is exchanged can be extended to new types of information (cf. Sect. 2.2)

3. We want to create a new format for verification witnesses that is more extensible, has clearer semantics, is more readable and concise than the current, GraphML-based format (cf. Sect. 2.3)

Let us now look into these objectives with a bit more detail:

1. Leverage the Current Exchange Format. We want to determine whether new tools built using the existing exchange format for interchange of information have the potential to improve upon the state-of-the-art in automatic software verification and validation. Furthermore, we want to investigate whether the exchange format is also useful for different domains than automatic software verification, namely we want to investigate whether interactive verifiers can benefit from the information provided by automatic verifiers and vice versa.

2. Improve Information That Is Exchanged. In the landscape of software verification, the content of the information exchanged plays a crucial role in validating and verifying software. Currently, witnesses primarily contain invariants as C expressions. However, this limits the expressiveness to a point where complete (e.g. inductive) proofs can sometimes not even be expressed in the GraphML-based format. Our objective is to explore new types of information that can both enhance verification and validation.

3. Improving the Exchange Format. In its current form, the GraphML-based witness format has the disadvantage that many verifiers choose to simply output a minimal witness that does not contain actual information [6], presumably because adding full support of the format is hard. Our aim is to introduce a new format that addresses these issues, making it easier for developers of automatic verifiers to export their information, in the way they see best fits their verification approach. The envisioned format shall be more intuitive, extensible, and equipped with clear semantics. In addition, it should be designed to be more readable and concise, facilitating easier parsing and comprehension. A standardized format with well-defined semantics can help ensure the completeness and relevance of the information shared.

1.3 Structure

With these objectives outlined and detailed, the subsequent chapters will delve deeper into methodologies, experiments, and results to support the pursuit of the aforementioned objectives. The overarching goal is to push the boundaries of software verification, aiming for more rigorous, transparent, and collaborative approaches that can cater to the diverse challenges posed by modern software systems. In Chapter 2 we give a high-level overview of how our contribution fits into the world of software verification by describing the various approaches with their input/output characteristics as transformers. Afterwards, we take a step back and describe the research method that we employ in our publications in Chapter 3, covering the importance of formulating hypotheses and following good practices of open science. Chapter 4 explains the manuscripts that comprise this thesis in more detail. Finally, in Chapter 5 we summarize our findings in relation to the formulated research objectives and give an outlook on the future work that is now enabled by our contributions.

2. Leveraging Invariant Information

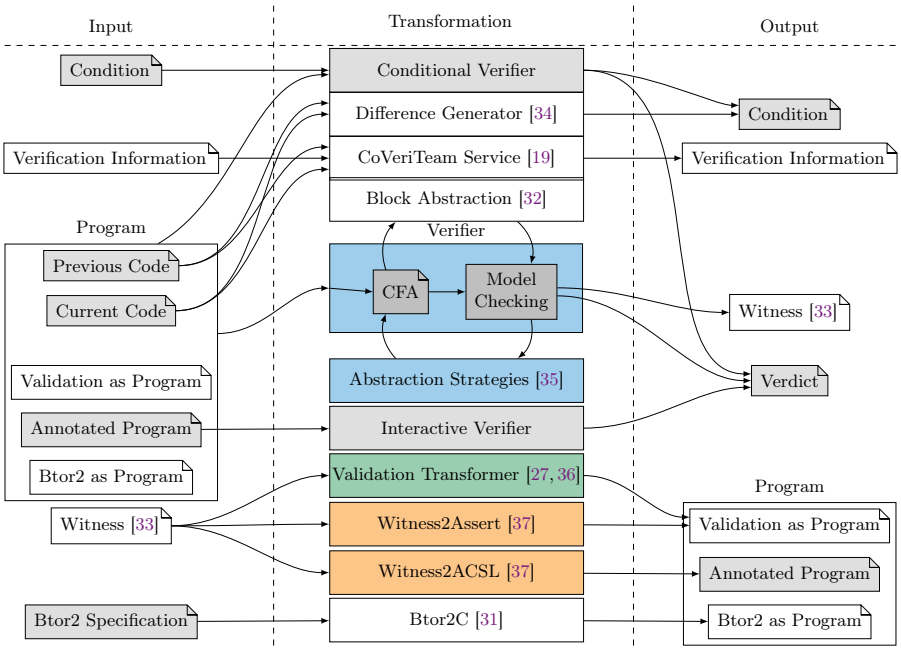


Figure 2.1: Overview of various transformations in software verification; highlighted in color are the contributions of this thesis [26]

The classical description of the automatic verification problem is that given a program and a specification the verifier has to compute the so-called verdict, i.e., answer the question whether the property given by the specification is fulfilled by the program.

In practice, this verification problem is undecidable, hence it is not guaranteed that the answer can be computed no matter how much time we allow for the computation, i.e., verification will be a semi-algorithm. In order to get meaningful results, one generally limits the resources, e.g., sets a timeout, such that the semi-algorithm can be turned into an algorithm. As a result, the allowed verdicts are either true if the program was proven correct with respect to the specification, false if a bug was found, or unknown in case the computation was terminated before a definite answer was found.

This automatic approach has the downside of trust in the computation result mentioned in Sect. 1.1. After all, there is just the verdict, no artifacts beside this to really understand the nature of the computation that was performed. In order to improve upon this, we need at least one additional output next to the verdict, which allows us to –(either manually or) with the help of a software system – *validate* whether the verdict can indeed be trusted. We refer to this kind of artifact as *witness*, and to the system that performs the validation as *validator*. Furthermore, we can distinguish between correctness validation, which is concerned with re-establishing proofs

from a verifier, and violation validation, which is concerned with re-establishing a concrete counterexample path to the specified property.

It would not make much sense to perform the validation with the very same algorithmic components that generated the verdict, as errors in these components would likely not be spotted. This is a bit like when we proofread text that we wrote ourselves. In case we learned the spelling of a word incorrectly, we would not spot this word as containing a misspelling, no matter how often we would proofread our own text. The upside is that for someone else, the wrong spelling will usually be obvious. If both persons do not spot the wrongly spelled word, this means it is a common mistake, which unfortunately exist in the English language as well as in the interpretation of the semantics of computer programs. An example would be whether a term like `(INT_MAX+1)*0` contains an integer overflow in the C programming language. These edge cases of program language semantics are also often not very clearly specified by the language specification.

Our hope is that if we use many different validators, we increase the chance to spot these cases that turn out to be common pitfalls. This is why it is generally desirable to perform validation with as many validators as possible.

Previously we described verifiers as (semi-)algorithms using their inputs and outputs to characterize what the verification problem is. We can now do the same for validation. A validator takes as input a program, a specification, and a witness, and returns a verdict. For the meaning of the verdict, we can simply reuse the verdict of the original verification to signal whether a witness was found to be correct. If the original verdict was reproduced, the witness is successfully validated.

In fact, we can go one step further and also let the validator output a *refined witness*, allowing us to validate the validator [14]. In the end what we have done is looking at the verifiers and validators from a point of view of *transformers*, taking their inputs and outputs and connecting them accordingly. A unifying component framework has been proposed [30] to emphasize this view on the various approaches that are used in formal methods, and to encourage the systematic exploration on how these can be used as building blocks to come up with novel approaches that are based on cooperation of these components.

In order to easily explore possible combinations of tools, we can draw artifacts as both inputs and outputs and then add transformers by connecting them to the corresponding input and output artifacts, as shown in Fig. 2.1. This way, once we applied a transformer on some inputs and generated an output, we can just look for that output in the input column and explore ways how this input can then be used in another transformation. As a side note, we leave out the specification in Fig. 2.1 to make it more readable. The specification is usually an input for all the transformers, and sometimes one transformer might choose to change the specification when its artifacts are used by the next transformer, but these are details of the transformers for which we can always consult the corresponding publications if interested.

As example, we can take conditional verification [16]. The idea behind conditional verification is to instead of returning the verdict unknown in case the resources are exceeded, we at least produce an artifact, the condition, that marks the parts of the state space that have already been proven correct. This way, we can run another conditional verifier that accepts a condition as input, and which then only tries to verify those parts of the state space that are left to explore by the condition.

A similar approach can be done if we look at two slightly different versions of a program. This would for example be useful in cases where a piece of software is developed over time in small increments, and we do not want to verify the whole program from scratch every time. By identifying the parts of the program that actually need to be verified again using a difference generator [34], we can generate a condition and hand the task of verifying this changed part to a conditional verifier.

When comparing the signatures of a verifier and validator, we notice that both transformers look very similar. The validator is, in essence, a verifier that has an additional obligation to take into account. For violation witnesses, it has to restrict the state space according to the information given in the witness and check whether the specification violation occurs at the part of the state space that the witness designates. For correctness witnesses, it has to verify the original program and in addition check whether the provided invariants are correct. A potential benefit here is that checking invariants is often easier than synthesizing the right invariants from scratch.

Motivated by the fact that verification and validation are so similar, and by the approach to chain verification tools like conditional verifiers using a shared exchange format, the idea developed to transform the validation task into a verification task, allowing us to use off-the-shelf verifiers for validation.

2.1 Exploring Cooperation via the Existing Exchange Format

A first implementation of this approach was MetaVal [26], which for violation witnesses is similar to reducer-based conditional model checking [17], interpreting the protocol automaton from the witness as a condition. For correctness witnesses however, instead of reducing the program, we actually add additional assertions to also check for the invariants from the witness. Technically, in both cases, we construct a product of the control-flow automaton of the original program and the protocol automaton of the witness. This product automaton is then converted into a C program. In Fig. 2.1 this is shown in the transformation block labelled Validation Transformer. MetaVal consists of such a validation transformer as described above and then hands the generated C program over to an off-the-shelf verifier for verification.

While MetaVal showed that this approach works and can in principle provide us with a wide range of validators, there are also some downsides to this approach. One problem that is specific to the way how the transformation via the automaton product works is that the resulting program is generally larger and not as nice to read by a human. While this might not be a problem for software verifiers, fixing bugs in the transformation or understanding why the validation fails becomes practically infeasible.

While it may seem like we reduced technology bias, we actually still heavily depend on the implementation of the program transformation. In fact, the validation transformer inside MetaVal is just one concrete instantiation of such a transformer. In practice one could even consider a validation transformer that breaks the validation task up into multiple programs. This way one can pinpoint why the validation fails.

This idea led to the design of LIV [27]. Much like a deductive verifier, we split up the verification task into multiple verification conditions along the invariants that are provided in the witness. This is similar to how a human would construct a Hoare-style proof for the program. The verification conditions, however, are transformed into C programs by LIV, such that they can be verified by existing, off-the-shelf verifiers. If we demand that the provided invariants suffice for an inductive proof, these C programs will not contain any loops, so we refer to them as *straight-line programs*.

Of course the additional requirement of inductive invariants is not something that is required of the current witness format, but demanding this is something that would make the problem of correctness witness validation actually decidable, provided the logic used to encode the straight-line programs' semantics is also decidable.

While LIV in its current form only supports the specification of loop invariants which can be expressed in the C language, this approach can easily be extended to also cover other proof information like function contracts and therefore allowing for a modular, inter-procedural scheme.

However, when the information cannot be expressed efficiently in the C language anymore, as would for example be the case for quantified invariants, it would be beneficial if the automatic verification community extended their tools to add special functions to add support for this. This could be incentivized by adding new categories to the Competition on Software Verification (SV-COMP), e.g., a category that provides support for quantifiers in assertions and assumptions.

2.2 Improving Information That Is Exchanged

The description in the previous section is naturally limited by what can be expressed in the established witness format. Up to now, the format for correctness witnesses only supports C expressions (without side effects) as invariants, that is, terms that are valid expressions in a C program at the location they refer to. It becomes quote obvious that this is not enough for a complete, concise correctness proof when looking at deductive verifiers and what they need to specify in order to prove more complicated programs.

Improving Exchange via ACSL. A good example to look at is the ANSI/C Specification Language (ACSL) [4]. This specification language is used by the interactive verifier Frama-C, but it is designed with a detailed documentation and formal semantics, ideal to borrow from for automatic software verification.

For specifying proofs for automatic software verification with witnesses in their current form, things get complicated for example when unbounded data structures are involved, that would require invariants with quantifiers. Another area of consideration would be function calls. While we can in principle inline non-recursive function calls, we would ideally want to have contracts with pre- and post-conditions. For termination, we ideally want to have variants, and the possibility to refer to program state at different locations, for example, not only at the current location, but also at the end of a loop.

This shows that both communities probably can benefit from cooperation. Automatic verifiers might be able to come up with invariants that could then be used in interactive verification, and vice versa. Using our framework of transformations, we can identify which transformations are necessary to allow to bridge the gap between both communities. To that end, we construct two transformers ACSL2WIT and WIT2ACSL which translate loop invariants annotated in ACSL into correctness witnesses and vice versa. This enables the construction of a lot of combinations, for example we can use an interactive verifiers like Frama-C as validator, or we can convert correctness witnesses into annotations that are easily readable by verification engineers and which allows them to use this as a starting point for interactive verification.

The approach of reusing software verifiers as tools for other verification problems (that they were not designed for initially) also inspired other approaches, not just the translation from and to ACSL. For example, Btor2C tries to bridge the gap between hardware and software verification by converting a hardware verification task written in Btor2 format into a software verification task that can be solved by an off-the-shelf software verifier.

Generalizing Towards Proof Strategies. While taking inspiration from ACSL for new types of information to track is interesting, this is in a way only declarative, not constructive. This means it does not directly describe the way how the proof is carried out. In theorem proving there are usual several proof strategies that can be tried out. The same is true for software verification. There are several optimization strategies that can be employed and that – sometimes even drastically – have an impact on whether a proof can be established or not. To a human these are often very clear, for example, sometimes exact information about what happens inside a loop is not necessary, just the information that the loop condition does not hold anymore after the loop is needed in order to prove the program correct. While this is often immediately obvious to

a human, many state-of-the-art analysis methods like predicate abstraction are prone to attempt to unroll this loop in case the crucial information is not determined right away. Strategies to abstract loops in order to overcome this are generally referred to as *loop abstractions* [45, 46], while approaches that precisely express the result of a loop are referred to as *loop accelerations* [54], which we treat as a special case of loop abstraction.

Loop abstraction strategies are one nice example of proof strategies that cannot be expressed in the GraphML-based exchange format. Usually, loop abstractions are implemented by program transformation of the input program, so the verification engineer has to decide which of these abstraction strategies to apply before handing the transformed program to the verifier or make an automated choice. We investigate whether loop abstractions suitable for a proof can be determined automatically using counterexample-guided abstraction refinement (CEGAR).[1] Our approach is depicted in Fig. 2.1 with the blue box titled "Abstraction Strategies" that interacts with the CFA generation and the actual model checking routine of the verifier (which in our case is CPAchecker). We encode different abstraction strategies directly into the CFA of the program, and modify the model checking algorithm to transparently hide all except the currently selected loop abstraction strategies from the actual analysis. The set of allowed strategies is tracked as so-called precision along the abstract states, similar to how predicate abstraction [2] tracks the current set of predicates. This has the advantage that the approach works transparently on different analysis domains, and can even be combined with regular CEGAR e.g. for predicate abstraction.

2.3 Improving the Exchange Format

In this section, we will first have a look at the existing exchange format and shed light on the areas where we see potential for improvement. Consecutively, we will present our proposal for our new, improved format. Lastly, we showcase potential of the new format with an application towards decidable witness validation.

Established GraphML-Based Format. The established, GraphML-based format [14] for correctness and violation witnesses has improved the trust in the verification results significantly [5]. However, as with every format, there are also some downsides which are hard to fix after such a format is established.

While it is in principle extensible, so far there had not been any major extension to the format. The format started out with the goal of providing witnesses for violation [15]. For this task, protocol automata are a nice way to describe the problem. Initial tool support included CPAchecker and UAutomizer, and both tools use automata internally, so the GraphML-based violation witness format was perceived as an encoding that could fit both CPAchecker's internal specification automata (which are based on the BLAST query language [12]) as well as UAutomizer's Floyd-Hoare automata. One of the major downsides here is that the semantics of the resulting automata is highly dependent on the choice of how to construct the CFA. Since both CPAchecker and UAutomizer use the EclipseCDT parser [61] for parsing the input C programs, their choices in constructing the CFA often match. However, this might not always hold for other tools that want to provide validation capabilities or even just generate violation witnesses that can be understood by one of the established validators.

Later on, GraphML-based witnesses for correctness were introduced [13]. Since the format that encodes protocol automata already existed, the decision was taken to also encode witnesses for correctness using the same automata. In order to achieve this, the kind of protocol automata

[1] A discussion of the corresponding publication can be found in Sect. 4.6.

were limited to observer automata, i.e., they were not allowed to restrict the state space in any way. A mechanism was added that allowed certain automaton states to be annotated with an invariant that should hold as long as the automaton is in this state. This can then be used by the validators, with the main use case being the extraction of loop invariants for re-establishing the proof of the original verifier. In theory, the rich automata language would allow to encode more complicated invariants into a correctness witness.

For example, an invariant that holds on every second time when entering a loop could be easily expressed. This expressiveness is so far however not really exploited by any of the verifiers and validators, and comes at the cost that correctness witnesses are far less readable and more prone to errors in the matching of an invariant to the right location. Just to give an example for the latter, the CPAchecker-based correctness witness validator allows an invariant to be matched to multiple locations, and only if all of them are ruled out as locations where the provided invariant holds, it is rejecting the witness [13]. As a consequence of these design decisions, correctness witnesses can be considered to represent partial proofs, and many verifiers opt to just generate an essentially empty correctness witness, as this is always guaranteed to be a correct witness in case the program indeed fulfills the specification. At first glance this might sound useless, but a validator might still be able to confirm the proof even in the case when the witness is empty. This will lead to the validator actually performing the complete verification on its own. A similar shortcoming can be observed for the violation witnesses.

New Format for Verification Witness. Because of the aforementioned points, we saw the need to propose a new format that solves the most-pressing issues with the old format.[2] Instead of GraphML, which is a format that expresses graphs as XML, we use YAML as format, which is both easily readable by humans and computer systems alike.

For correctness witnesses, our approach is an easy and great simplification of the existing format. We provide a way to store location invariants or loop invariants in separate entries. Each of these entries maps the invariant to its location in the source code. A correctness witness in the YAML format then consists of a (potentially empty) array of these entries. Each entry also contains meta data about the verification task, the producer of the witness, and other important information that is also present in the GraphML-based format. In case a verifier generates multiple invariants for a program, this metadata would be highly redundant, so we offer a way to specify a set of invariants that share the same metadata, improving readability and size of the witnesses. Regarding expressiveness, one would assume that the new format loses generality over the old format, which might even be the case, but does not play a role for the current validators. In our evaluation, we were able to show that correctness witness validation using the new format achieves practically identical performance to the old format. As explained before, current validators do only use the GraphML-based format to extract location and loop invariants. We argue that more complicated invariants, e.g. those that would only hold at each other visit of a loop location, is hard for a human to understand and could either be converted into a real invariant that always holds at this location or will turn out to not be useful for most validation approaches anyway.

For violation witnesses, we want to mainly address the problem of the semantics determining on the choice of the CFA of the program. While the GraphML-based violation witnesses focus on transitions in the CFA, we focus on sequence points in the program's AST. In a way, our violation witness specification fits better with the actual language semantics as defined by the C standard [51], since we focus on only using concepts that are also found there. Instead of allowing arbitrary automata, our violation witnesses are constructed as an ordered list of so-called segments, while each segment consistis of one follow waypoint and optionally as many avoid waypoints as required. The last segment ends in a special follow waypoint that marks the violation

[2] A discussion of the corresponding publication can be found in Sect. 4.6.

as being reached. The follow waypoints can be thought of as a similar feature to breakpoints in debugging. With each reached follow waypoint, we come one step closer to the violation. While in debugging, we cannot easily specify which paths not to follow, for verification we generally can give more abstract information. This is why we have the additional concept of avoid waypoints. Inside a segment, a validator shall disregard all paths through avoid waypoints. Similarly, we offer a way to attach assumptions to a follow waypoint. The validator only needs to keep those paths that fulfill the assumption. These assumptions shall hold at the sequence point that the waypoint points to, and not like with the GraphML witnesses after the next transition, because the transition concept is not part of the C standard, while sequence points are. For our first iteration of the new YAML-based format for violation witnesses, we limit allowed sequence points to those at the beginning of full expressions, expression statements, as well as those before a function is entered and after it is exited. In the future, we might support more sequence points, but this would also make adding support for the new format more complex, and might actually not be needed in terms of expressiveness, much like with a debugger one usually cannot add breakpoints at every location one would like, but it is usually precise enough to just specify a certain line or statement in the execution. The new violation witness format is easier to read for humans and also smaller in size, though of the reduction is not as drastic as with the correctness witnesses, as we still need to encode information that is proportional in size to the length of the counterexample path that would also be present in the GraphML-based format.

Decidable Witness Validation. So far, the new format does not address the problem that validation is still undecidable and therefore one cannot expect validation to terminate quickly, very much in contrast to how a proof would be checked inside an interactive verifier. Once all the information for the proof is present, checking the proof should not be as complicated as coming up with the proof. A similar argument can be made for violation witnesses, as recently logics that make it possible to prove the presence of bugs, like the infamous incorrectness logic [59] have been proposed [44, 68].

We will look at correctness witnesses first, as the picture there is very clear. For correctness witnesses even in the new YAML-based format, an empty set of invariants is in principle also a valid correctness witness. This is something that can be addressed outside the exchange format, by defining a different semantics for the validation. We follow this idea by constructing a Hoare-style proof and demanding that all information (i.e., invariants) necessary for the proof is present. In other words, we pose the additional constraint that apart from being safe, meaning that the invariants rule out the presence of specification violations, the invariants shall also be inductive in relation to each other. We investigate this approach in a prototype tool called LIV.[3] While for the publication, LIV had to use the GraphML-based format, as the YAML-based exchange format was not yet published, LIV supports the YAML format from the start, as this is much simpler to support than the GraphML-based format. The support for the GraphML-based format is only very basic and is limited to extracting the invariant locations, otherwise ignoring the graph structure present in the witness. We are able to show that complete proofs can already be established with this approach for a significant fraction of the invariants currently produced by the verifiers. In LIV, we generate verification conditions from the proof obligations in the Hoare-style proof and convert them back into C programs, such that they can easily be checked by the very same verifiers that are producing the verification witnesses in the first place, greatly simplifying the complexity of the validation tool chain.

For violation witnesses, an approach that would allow checking of violations in a similar manner has not yet been proposed. In practice, this problem is also not that present as for correctness, since in case a violation was found, the error path is indeed finite, so also checking

[3] A discussion of the corresponding publication can be found in Sect. 4.5.

the counterexample is in general faster than the verification [15]. A verifier could decide to output a minimal violation witness, in which case the validator would need to explore the unrestricted state space and potentially run into a timeout. This is because in the Competition on Software Verification(SV-COMP) the available CPU time is usually reduced from 900 s to 90 s to incentivize verifiers to output meaningful violation witnesses.

As with correctness, our format does not contribute to improving this aspect of validation. In principle, there could be an unbounded loop unrolling between two waypoints in the new format, much like there could be cycles in the violation witness automaton of the GraphML-based format. One could pose additional constraints, e.g. that there are no infinite executions between two waypoints, but proving this would itself be as hard as the halting problem. Another option would be to demand that there is a waypoint for every time an iteration statement like a loop is visited, however this would negatively affect the size of a witness and force us to document each loop unrolling even for trivial loops. In order to avoid the shortcomings of these two options, actually encoding a proof of violation in the witness is a promising future research direction, and would require to prove the termination of loops e.g. using a loop variant. Information like loop variants can more naturally be expressed in the YAML format for correctness witnesses. We currently plan to look into incorrectness logic [59] to achieve this as part of our future work.

3. Research Method

"The method of science, as stodgy and grumpy as it may seem,
is far more important than the findings of science."

– Carl Sagan, 1995 [62]

Apart from the scientific findings that we presented in the previous chapter, it is also worthwhile to look at the scientific method that we employ. After all, it is clear that an ill-intended researcher could almost always tailor their results in a way to support the outcome they want to present. Such a behavior would of course be highly unethical, but it does not always require ill intent to draw scientific conclusions that are far off. For example, there could be unconscious biases or decisions that influence the outcome of experiments in a way that lacks scientific scrutiny.

3.1 Hypothesis-Driven Approach

Sometimes, we can observe publications that present an interesting approach, but it is unclear what the actual contributions are. Even if the contributions are stated, it might remain unclear to which extent the performed experiments – if any – back up the claimed contributions.

To avoid this problem, we always perform experiments and formulate research questions beforehand for all our publications, posting hypotheses that can be quantified and clearly decided. We use reliable benchmarking [23] via BenchExec to make sure the experiments are performed to the highest standards of reproducibility. BenchExec makes use of certain features from the Linux kernel such as namespaces and cgroups to ensure that resource consumption is reliably monitored and restricted to the specified limits. For benchmarks we mainly concentrate on an established set of C programs [11]. These programs are curated by the community of the Competition on Software Verification (SV-COMP), in a way that ensures that a consensus exists about their semantics.

3.2 Open Science

"Arts and sciences, research and teaching shall be free."

– Article 5 (3) of the German Grundgesetz

The above quote from the German constitution makes it clear that it is desirable to grant science special freedoms. In addition, our scientific research is usually paid for by public funds. As such, it seems imperative that we also allow the public to access all our findings as freely as possible.

We achieve this by making our publications openly accessible, together with a reproduction artifact. In addition, the software where we implement our approaches is licensed under a permissive open-source license. Table 3.1 lists the corresponding open-source software and reproduction

Publication	Open-Source Software	Artifact DOI
MetaVal (Sect. 4.1)	gitlab.com/sosy-lab/software/metaval	10.5281/zenodo.3831417
Frama-C Wrapper (Sect. 4.2)	gitlab.com/sosy-lab/software/frama-c-sv	10.5281/zenodo.5959149
Loop Abstraction (Sect. 4.3)	gitlab.com/sosy-lab/software/cpachecker	10.5281/zenodo.6793834
ACSL (Sect. 4.4)	gitlab.com/sosy-lab/software/cpachecker	10.5281/zenodo.6541544
LIV (Sect. 4.5)	gitlab.com/sosy-lab/software/liv	10.5281/zenodo.8289101
YAML Witnesses (Sect. 4.6)	gitlab.com/sosy-lab/software/cpachecker[1]	10.5281/zenodo.10826204

Table 3.1: Overview of publications with openly accessible software and reproduction artifacts

artifact for each of the publications presented in this thesis. In the following, we will explain the different aspects of open science and give some context as to why this is important.

Open Source Software. All components that were created as part of this thesis are available as open-source software under a permissive license (Apache 2.0). This is important because it allows other researchers to check the implementations for potential errors and bugs, but also to build upon existing work and compare to the state-of-the-art. We found during our own research that more than often, a fair comparison with the state-of-the art would be desirable, but is not possible because the corresponding software is not under a permissive license that allows at least execution in a scientific context. Sometimes it is even the case that existing implementations are not available anymore.

In order to improve the situation for researchers that want to inspect or build upon our work, we follow good practice and publish an artifact along with every publication.

Open Reproduction Artifacts. A reproduction artifact is a collection or package of materials, data, code, and documentation, that allows for the consistent reproduction and thus validation of experimental results.

In order to guarantee that the results can be reproduced on the long run, ideally the artifact is self-contained. For that reason, virtual machine or container images are often a good choice. Inside these containers, the complete execution environment and dependencies as well as additional documentation can be stored such that it will be executable as long as a suitable hypervisor or operating system to run the images on is available.

In order to ensure that the artifacts will not eventually vanish, we cannot trust privately held companies like github or gitlab, even established services might eventually end up being taken down. To just name one example, Google Code stopped its service on January 25th, 2016, only 10 years after the service's initial launch.[2] For this reason, we choose to publish artifacts on Zenodo, an open repository initiated by the European OpenAIRE program and operated by CERN. According to Zenodo's General Policies v1.0[3], data will be retained for the lifetime of the (Zenodo) repository, in other words, it is currently planned that older data on Zenodo will not expire, at least not as long as Zenodo exists. The existence of the Zenodo repository itself is ensured by the current host laboratory (CERN), which already has a planning horizon spanning the next 20 years.

While gathering experience with creating reproduction artifacts, we discovered the need to make the creation of reproduction artifacts itself reproducible. Usually, there are very specific steps required to compile an artifact, with all its dependencies in such a way that the final artifact

[1] CPAchecker is listed here as an example, for a full list of open-source software related to the YAML witnesses, have a look at the list of tools in the Zenodo artifact

[2] https://opensource.googleblog.com/2015/03/farewell-to-google-code.html, accessed on 2023-10-01

[3] https://about.zenodo.org/policies/, accessed on 2024-12-03

will work in isolation. This can be a tedious process, and the knowledge how to create the artifact can be lost very quickly, even for the individual that created the artifact in the first place.

As a result of this, we developed a process of continuous artifact creation, where the artifact creation is itself completely tracked and versioned. All steps necessary to prepare and execute the experiment are tracked in a Makefile [64]. In case an artifact VM is desired, we use Vagrant [60] and its so-called Vagrantfile for managing the VM creation and export. The repository for the artifact together with the Makefile is copied inside the virtual machine, from where necessary dependencies can be resolved as set up in the Makefile. An example can be found in the repository for the artifact of the publication about Loop Abstractions (Sect. 4.3)[4] or LIV (Sect. 4.5)[5].

Open Access Publications. It is important for other researchers to be able to freely access our publications. Otherwise, especially people that are not backed by a financially robust institution might be cut off from the newest achievements, hindering their ability to contribute to scientific progress. Fortunately, it is generally possible to negotiate open access conditions with a publisher either upon publication or even at a later time after publication.

We are glad to be able to state that all publications in this dissertation are available under open access conditions. This has several advantages, both of altruistic and self-interest-driven nature. On the one hand, it levels the playing field for other researchers, not discriminating researchers whose institution cannot pay for access to a certain journal. On the other hand, it also improves the chance of our work being discovered, read, and cited by other researchers.

[4] https://gitlab.com/sosy-lab/research/data/loop-abstraction
[5] https://gitlab.com/sosy-lab/research/data/liv

4. Discussion of Manuscripts

In this chapter, we briefly discuss each of the publications that are attached to this thesis in chronological order. We briefly explain the problem and background behind each publication followed by a description of the approach and experimental findings determined in the evaluation

4.1 MetaVal: Witness Validation via Verification

The article *MetaVal: Witness Validation via Verification*, which is reprinted in Appendix A, pages 42–54 of this dissertation, was authored by Dirk Beyer and Martin Spiessl, and published by Springer in the Proceedings of CAV 2020, pages 165–177 [26].

MetaVal is the first approach that leverages off-the-shelf verifiers for witness validation, and allows to use verifiers that understand the verification task conventions of the Competition on Software Verification [7] to be used for validation. This helps to reduce technology bias, especially on the side of correctness-witness validation, where only two tools (CPAchecker and UAutomizer) existed before MetaVal was presented. That being said, there is still some technology bias towards CPAchecker, as this is the tool used to perform the transformation from a validation task into a verification task.

This works by constructing a product automaton between the CFA of the input program and the witness automaton. The resulting automaton is mapped back to a CFA, which is ultimately converted into a C program, similar to what is done in reducer-based conditional model checking [17]. For correctness witnesses, the witness automaton has to be modified slightly from the original version [13] in order to express that the invariants shall not be violated.

In the evaluation, MetaVal is executed on witnesses from SV-COMP 2020. As backend verifiers, CPAchecker, UAutomizer, and Symbiotic were selected. The result shows that MetaVal can complement existing validators, especially when it comes to correctness-witness validation, where there are only the aforementioned two other tools.

The concrete implementation of the transformer in MetaVal is not ideal, for the following two reasons. Firstly, the generated programs are hard to read, making inspection of failed validation attempts by a user very hard and sometimes practically impossible. Secondly, even if the generated program were simple enough, there is no guarantee that the employed transformation via the product automaton is indeed soundly encoding the validation problem. Neither the implementation of the transformation nor an abstract description thereof has been formally proven to be correct. These are however shortcomings of the implementation, and could be addressed by providing a different transformer with an easier transformation and additional soundness proofs. Currently, the transformer is based on CPAchecker, which introduces additional tool bias. Ideally, an improved transformer comes as a separate tool that does not bias towards a specific verifier.

A reproduction package is available which makes it possible to reproduce all results from the article [25].

Martin Spiessl is the main author of the article. His contributions are (1) the formalization of the approach in terms of an automaton product between the CFA of the input program and the witness automaton, (2) the development of the presented tool MetaVal, and (3) drafting of the motivating example together with the illustrations.

4.2 The Static Analyzer Frama-C in SV-COMP (Competition Contribution)

The article *The Static Analyzer Frama-C in SV-COMP (Competition Contribution)*, which is printed in Appendix A, pages 55-60 of this dissertation, was authored by Dirk Beyer and Martin Spiessl, and published by Springer in the Proceedings of TACAS 2022, pages 429-434.

The article is about a wrapper around Frama-C called Frama-C-SV that allows this interactive verifier to analyze verification tasks from the SV-Benchmarks set and therefore participate in the Competition on Software Verification(SV-COMP).

In order to achieve this, the SV-COMP specific conventions are adapted in such a way that they can be understood by Frama-C by a so-called input transformer. For example, a harness is provided along the original program file that implements the SV-COMP specific functions and expresses them in terms of special functions used by Frama-C. Another important step is to interpret the results of Frama-C with a so-called output transformer. Since a static analyzer in general cannot prove the definite presence of specification violations, we concentrate on cases where Frama-C proves the input program correct, focusing on soundness over number of proofs, since Frama-C is considered a sound static analyzer [38]. Since SV-COMP requires the generation of correctness witnesses for proofs, we also generate a minimal correctness witness along with the verdict, as many other participating verifiers do. Exporting useful information like found invariants is still something that is planned to be investigated as part of future work.

As Frama-C provides several different analyses and configuration options influencing the performance, part of this work consists in making a suitable choice and documenting the findings. We decide to use the EVA analysis [40], a value analysis with tuneable precision, over the weakest precondition module, as the latter would require extensive annotations to the input program in order to succeed with any proof. While we choose a certain set of options for SV-COMP, Frama-C-SV and its tool-info module inside of BenchExec are designed in such a way that these options can easily be adapted depending on the property that shall be verified, allowing other researchers to easily perform experiments with a different set of options.

The results for Frama-C in the 11th Competition on Software Verification (SV-COMP 2022) [9, 10] show that it is indeed possible for a static analyzer to successfully participate. However, as a static analyzer is designed to be fast and work on large codebases, Frama-C generally finishes quickly and thus does not take full advantage of the available CPU time per verification task. Another way to look at this is that Frama-C could most-likely scale to verification tasks that far exceed the size of the benchmarks currently present in the SV-Benchmarks.

Martin Spiessl is the main author of the article. His contributions are (1) implementation, technical conception and optimization of Frama-C-SV for the SV-COMP-specific requirements and (2) description of the approach in the competition contribution publication.

4.3 A Unifying Approach for Control-Flow-Based Loop Abstraction

The article *A Unifying Approach for Control-Flow-Based Loop Abstraction*, which is reprinted in Appendix A, pages 79–95 of this dissertation, was authored by Dirk Beyer, Marian Lingsch-Rosenfeld, and Martin Spiessl, and published by Springer in the Proceedings of SEFM 2022, pages 3–19 [35].

The article describes a novel approach to explore different combinations of loop abstraction techniques using a CEGAR [41] pattern. Verification of loops poses one of the main challenges in program verification, and in many cases, it is possible to replace a loop by a much simpler

but over-approximative version that still is sufficiently precise to allow for proving the specified property. So far, approaches applied the most promising of these loop abstraction strategies as a program transformation before the actual verification [1].

The novel approach in this article weaves multiple loop abstraction strategies transparently into the CFA of the input program. Along with the abstract states of the actual analysis that performs the state space exploration, we track for each loop location the set of available loop abstractions (the precision) and limit the transitions of the analysis to only allow one version at a time. In case a counterexample is found that contains on its path abstract states belonging to a strategy that is over-approximating, we consider the counterexample to be spurious and exclude that strategy from the precision and recompute the reachability graph using the next best strategy that can be applied. If all over-approximating strategies have been ruled out, the analysis defaults back to the original version of the loop, so in the end, the approach can solve any task that the unmodified verifier can solve. The overhead of checking the loop abstractions is usually negligible, since loop abstractions remove loops from the program and therefore either succeed or fail quickly in general.

We implement our approach in the software verification framework CPAchecker. While we concentrate on four basic loop abstractions, the approach can easily be extended with more complicated loop abstractions. As an edge case we also include constant propagation, where loops containing linear arithmetics with easily computable results are replaced by assignments of the final values to the corresponding variables. This is a precise acceleration strategy, so counterexamples found with this strategy applied are actually considered to be concretizable. In order to increase reusability of the generated abstractions and for determination of the external validity we make the successful loop abstractions available in the form of program patches.

In the evaluation, we are able to show that our CEGAR-based loop abstraction approach can indeed solve more verification tasks than the plain analyses inside CPAchecker, while only requiring an acceptable, small overhead in computation time. We further show that for those cases where we solve additional tasks, also other verifiers like Cbmc can benefit from the abstractions that we export as patches, increasing the trust in our verification procedure. Indeed, bounded model checking approaches (Cbmc and the CPAchecker-internal BMC approach) seem to especially benefit from loop abstractions, which is not surprising, given that this enables them to prove programs correct that would otherwise be unrolled indefinitely.

A reproduction package is available which makes it possible to reproduce all results from the article [21].

Martin Spiessl and Marian Lingsch-Rosenfeld are the main authors of this article. Martin Spiessl's contributions are (1) theoretical description of the approach including the Hoare proofs for selected loop abstraction strategies, (2) generation of patches from the abstractions in the implementation, and (3) conduction of the experiments and creation of the artifact in a reproducible way.

4.4 Cooperation between Automatic and Interactive Software Verifiers

The article *Cooperation between Automatic and Interactive Software Verifiers*, which is reprinted in Appendix A, pages 61–78 of this dissertation, was authored by Dirk Beyer, Martin Spiessl, and Sven Umbricht, and published by Springer in the Proceedings of SEFM 2022, pages 111–128 [37].

The article describes an approach that transfers information between automatic and interactive software verifiers. Automatic verifiers are those that do not require input from the user (despite initial instrumentation of the verification task) and try to come up with a proof or counterexample

of the specified property automatically. Examples include well-known tools participating in the Competition on Software Verification (SV-COMP) such as Cbmc [42], CPAchecker [20], or UAutomizer [50].

In contrast, interactive (or sometimes also referred to as deductive) verifiers are built in such a way that they require frequent interaction with the user, e.g., to provide information such as loop invariants, function contracts, auxiliary predicates. Examples include well-known state-of-the-art static analyzers like Frama-C [43], VeriFast [52], or VerCors [39].

Until the publication of this article, the communities around both of these approaches acted separately, and the question whether cooperation could be achieved was still open. To change this, we focus on Frama-C as example for the interactive verifiers, because it features a well-documented specification language called ACSL [4] (which is based on JML [56]). For the automatic verifiers, we use the correctness witness format that the automatic verifiers participating in SV-COMP support, allowing us to use a wide variety of tools. We implement transformers between these two formats and construct new tool combinations using an approach inspired by the unifying component framework for cooperative verification [30].

We are able to show that Frama-C can reuse information present in the verification witnesses from automatic verifiers and prove more verification tasks correct than without this information. Moving into the other direction, we can show that ACSL annotations crafted by users can successfully be translated into GraphML-based witnesses, which can help a validator establish a proof. Since the number of automatic validators is far lower than the number of verifiers, we implement a different component that turns ACSL annotations into plain assertions inside the input program and evaluate how this affects verification with automatic verifiers.

A reproduction package is available which makes it possible to reproduce all results from the article [28].

Martin Spiessl is the main author of the article. His contributions are (1) preparation of the evaluation and artifact in a reproducible fashion (2) instrumentation of Frama-C for enabling its use in the evaluation (3) conceptual work on the approach (together with the co-authors).

4.5 LIV: Invariant Validation using Straight-Line Programs

The article *LIV: Invariant Validation using Straight-Line Programs*, which is reprinted in Appendix A, pages 96–99 of this dissertation, was authored by Dirk Beyer and Martin Spiessl, and is published by Springer in the Proceedings of ASE 2023.

The article describes a new correctness-witness verifier called LIV which is inspired by Hoare-style proofs. The main idea is to demand that the invariants in the witness are sufficient to establish a full proof, then slice the program into several C programs (so-called straight-line programs) that encode the verification conditions from the Hoare-style proof. These can then be validated by an off-the-shelf verifier.

Similar approaches like VST-A [66] already exist and show that the transformation into straight-line programs can be proven to be correct. However, with LIV we focus on the abstract syntax tree (AST) of the program, whereas in VST-A the actual transformation is still performed on the control-flow graph (CFG). Many other tools transform their verification conditions into an internal format [3, 43, 47, 48, 52, 57]. The novelty introduced in LIV is that we output the verification conditions in the same format as the input programs, leveraging the conventions of the Competition on Software Verification.

The main philosophy is that the splitting based on the AST is simple and prevents bugs that come from the exact implementation of the CFG. As such, a tool that performs the splitting does

not need to deal with all the exact semantics of a C program, it just needs to understand as much as to soundly split the program into conditions.

An additional benefit of the splitting is that it is easier to locate the reason as to why the validation fails, so users can get additional feedback upon validation. Since the generated straight-line programs do not contain loops, the validation task becomes decideable if the underlying theory is decideable. Since witnesses only support C expressions as invariants, the verification tasks corresponding to the straight-line programs are decideable. This might change in a future witness format that e.g. also supports quantified invariants. Also the support for further proof information like function contracts might further improve the usefulness of LIV.

For execution of the verifier backend, we use CoVeriTeam [18] as an abstraction layer to delegate verification of the straight-line programs to any of the verifiers participating in SV-COMP transparently, and benchmarks are executed using reliable benchmarking with BenchExec. In the experiments of the article, we show that existing witnesses from verifiers already contain invariants that are sufficient for an (inductive) proof, despite the fact that verifiers are not (yet) tuned to output inductive, safe invariants, as this is currently not a requirement. As benchmark set, we use a subset of the SV-Benchmarks which is annotated with supposedly inductive and safe invariants, and are able to successfully detect cases in which these annotations are not yet enough to establish a proof. Determining the reason because of which the proof fails is shown to be easy because of the decompositional approach of LIV.

A reproduction package is available which makes it possible to reproduce all results from the article [27].

Martin Spiessl is the main author of the article. His contributions are (1) formalization of the approach, (2) implementation of the tool, and (3) experimental evaluation and artifact creation.

4.6 Improved Witnesses for Software Verification

The article *Improved Witnesses for Software Verification*, which is reprinted in Appendix A, pages 100–119 of this dissertation, was authored by Paulína Ayaziová, Dirk Beyer, Marian Lingsch-Rosenfeld, Martin Spiessl, and Jan Strejček, and published by Springer in the Proceedings of SPIN 2024.

This article introduces a new format for verification witnesses that offers some improvements over the existing, GraphML-based format that encodes witnesses as protocol automata. This encoding into automata is based on CPAchecker's specification automata and UAutomizer's Floyd-Hoare-Automata, i.e., it is designed to capture their semantics well. However, this might hinder adoption by other tools that have a different interpretation of the program's CFA (or are not CFA-based at all).

One major drawback of this automata-based format is that the semantics is only specified with relation to the CFA, i.e., it is under-specified in certain edge cases where the meaning is different depending on how exactly the CFA looks. Since the C standard does not explicitly specify that there is a CFA or how it looks, the format proposed in this article focuses more on syntactical features of the C language and on features directly defined in the C standard, most importantly sequence points. These are points in between full expressions in the program at which all side effects have been resolved. As such, these are natural points in the execution of a program that are both easy for humans to grasp and for validators to reason about. While the GraphML-based format matches information to certain *transitions* in the CFA, the new format attaches this information to certain *states* in the program's execution.

The new format is based on YAML and designed to be more easily readable by humans.

For correctness witnesses, the GraphML-based format is in theory more powerful than the YAML-based one, this power was never actually used by the validators. This is why the new format which just stores the loop invariants together with their location instead of encoding this into an automaton is shown to closely reproduce the same validation results. As a side effect, the size of witnesses is reduced while without sacrificing validation power.

For violation witnesses, the new format defines a sequence of so-called waypoints that lead to the violation. This notion is designed to be intuitive and similar to how a human would use a debugger, setting breakpoints along the program execution until the desired program state is reached. Readability is increased, though witness sizes are not reduced in the same way as for correctness witnesses.

The new format is also designed with extensibility in mind. Every YAML witness consists of a sequence of entries that are of a specified entry type. If a verifier or validator developer wants to add a different kind of information that would be useful for validation, they can simply extend the format by a new entry type.

The implementation inside CPAchecker converts between the GraphML-based and YAML-based witnesses, further performance gains can be achieved in future work by leveraging the new format directly.

A reproduction package with DOI 10.5281/zenodo.10826204 is available which makes it possible to reproduce all results from the article.

All authors contributed equally to the text of the article. Martin Spiessl's contributions are (1) guiding the design of the new format (together with all co-authors), (2) implementation of the new format into CPAchecker (together with Marian Lingsch-Rosenfeld, who took care mainly of the support for correctness witnesses), and (3) creation of the experiment pipeline and artifacts in a reproducible fashion.

5. Conclusion and Future Directions

After having discussed all manuscripts that comprise this dissertation, we will summarize what has been achieved and give some outlook on the future work and logical next steps that became apparent during the current state.

5.1 Summary

We have seen that the existing exchange format allows the composition of novel tools and approaches, both inside the domain of automatic software verification, like in the case of MetaVal where we construct a validator from off-the-shelf verifiers, as well in discovering new approaches and bridging the gap to other areas of formal methods, like when converting between verification witnesses and ACSL annotations. We have also seen that there are some limitations posed by the format, which are improved by a new format, like a well-defined semantics, better readability, and overall size.

With LIV, we investigated whether a more strict correctness-witness validation approach can lead to better understandability of the validation results and make validation of correctness witnesses truly simpler and faster than verification. Regarding the kinds of information that can be exchanged between tools, we focused on loop abstraction strategies and were able to show that these proof strategies can be used in a CEGAR-style approach to guide the automatic verifiers towards a successful proof.

5.2 Future Directions

Despite the fact that we presented several novel applications of the existing exchange format and proposed a new, improved format, the presented publications only lay the groundwork for even more potential future work, which we want to quickly present in this section.

LIV and MetaVal: Towards a General Program Transformation Framework. In its current form, LIV is very simple and tailored towards one specific use case, namely (a more strict definition of) verification-witness validation. However, during the design, it became apparent that it would also be ideal as a framework to implement a variety of different program transformation approaches. For example, the MetaVal approach could easily be added to LIV. Instead of splitting the input program into multiple parts, we would just add assertions for the loop invariants there. To truly extend on this approach, the new semantics of the new segment-based violation witnesses could be soundly encoded as program transformation using some additional global variables to track the current segment. This is more straightforward as with the GraphML-based format, as waypoints are generally defined at positions (sequence points) where it is possible to insert instrumentation code.

Regarding loop abstractions, it seems natural to implement these program transformations into a tool like LIV that is focused on syntactic transformations over the input program. Indeed, it has been shown that the CEGAR-style approach of the unifying approach for control-flow-based loop abstractions [24] that we presented can also be constructed as a black-box approach [22] that

just works on patches generated for each loop abstraction application. The support of various program transformations would be possible in a similar fashion.

In order to achieve the above additions, we would need to develop LIV further past the current prototype stage to support a wide variety of C programs. For this, participation in Competition on Software Verification (SV-COMP) – either as a validator or together with an invariant generator even as verifier – would be an ideal way to achieve this, as the SV-Benchmarks used in SV-COMP cover a wide range of features of the C language.

Another direction that could be explored with LIV is to add support for other form of pre- and postconditions for the generated straight-line programs. So far, we rely on the conventions of SV-COMP and express these conditions in C, but other tools, especially interactive/deductive verifiers like VerCors [39] or VeriFast [52] feature their own annotation language, so these could be easily supported if pre- and postconditions were to be generated using these annotation languages.

More Strict Witness Validation. Recently, more effort has been put into giving incentive to authors of witness validators to improve their tools and to get more visibility and credit. For example, in SV-COMP 2023, a new validator track has been added where there is a score computation and medals for validators similar to the established verifier track [29]. While this is an effort that is long overdue, there are also some practical issues with this new addition. Because the semantics of the GraphML-based witnesses is not always clear, only witnesses that are generated with a verdict conflicting with the ground truth, i.e., the verdict of the verification task that has been assigned by the verification experts from the community, are used as invalid witnesses in the validator competition. But even if the witness and the verification task share the same verdict, a witness might be faulty and needs to be rejected, e.g. because it contains a wrong invariant. This case cannot currently be handled well, and the YAML witness format together with a more strict definition of witness validation might make it possible to address this problem in the future. Also, from a practical standpoint, looking at validator results for witnesses that can sometimes be several hundreds of MB in size is, simply put, infeasible for validator authors. With verification tasks, the tasks tend to be stable and not change too much over time. With verification witnesses however, these could change at any moment during the competition's preruns, making it especially hard for validator designers to check their results, and without any guarantee that the witnesses will be similar in the final run of the competition.

Support for New Kinds of Information. The new witness format laid the groundwork for extensions towards different kinds of information apart from invariants given as C expression. With loop abstractions, we presented only one type of such information, while we did not yet add this as extension to the new format. This seems like a logical first step towards this direction. Especially for verification of properties other than reachability, it seems natural that the need for other types of information will arise soon, especially in case the more strict witness validation approach will gain momentum, as there one needs to define what a full proof of correctness would look like. As an example, for memory safety, information like lock sets or happens-before relations might be something to investigate. Here a cooperation with the developers of the static analysis tool GobLint [65] is especially promising, since one of the strengths of GobLint is the verification of concurrent C programs and their tool cannot easily support violation witnesses in the way that the GraphML-based format intends, since they generally do not have full paths to counterexamples.

Continuous Artifacts. In Sect. 3.2 we briefly described the artifact setup pipeline that allows us to create an artifact VM for reproduction of our experimental results at any given time and in a reproducible manner. It would be interesting to investigate how this approach can be generalized and whether it is also useful to other researchers. Since experiments on software verification (especially if a large benchmark set like the SV-Benchmarks is used) tend to take up a significant amount of time, adding a way to transparently execute these experiments on a distributed cluster

of virtualized nodes would improve reproducibility of the experimental results. We currently use a custom-made system for distributing BenchExec run collections of our experiments on a cluster of compute nodes, the same system that is used to conduct the Competition on Software Verification [8]. Ideally, this system would be made available to run on a set of VMs that are described e.g. using Vagrant, in a similar manner as the artifact VM currently is for most of our artifacts.

Bibliography

[1] Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141. IEEE (2019). https://doi.org/10.1109/ASE.2019.00121

[2] Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. PLDI. pp. 203–213. ACM (2001). https://doi.org/10.1145/378795.378846

[3] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proc. FMCO. pp. 364–387. LNCS 4111, Springer (2005). https://doi.org/10.1007/11804192_17

[4] Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at `https://frama-c.com/download/acsl-1.17.pdf`

[5] Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS. pp. 331–349. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20

[6] Beyer, D.: A data set of program invariants and error paths. In: Proc. MSR. pp. 111–115. IEEE (2019). https://doi.org/10.1109/MSR.2019.00026

[7] Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21

[8] Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24

[9] Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20

[10] Beyer, D.: Results of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831008

[11] Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003

[12] Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast query language for software verification. In: Proc. SAS. pp. 2–18. LNCS 3148, Springer (2004). https://doi.org/10.1007/978-3-540-27864-1_2

[13] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351

[14] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579

[15] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867

[16] Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664

[17] Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259

[18] Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31

[19] Beyer, D., Kanav, S., Wachowitz, H.: CoVeriTeam service: Verification as a service. In: Proc. ICSE, companion. pp. 21–25. IEEE (2023). https://doi.org/10.1109/ICSE-Companion58688.2023.00017

[20] 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

[21] Beyer, D., Lingsch Rosenfeld, M., Spiessl, M.: Reproduction package for SEFM 2022 article 'A unifying approach for control-flow-based loop abstraction'. Zenodo (2022). https://doi.org/10.5281/zenodo.6793834

[22] Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: CEGAR-PT: A tool for abstraction by program transformation. In: Proc. ASE. pp. 2078–2081. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00215

[23] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y

[24] Beyer, D., Rosenfeld, M.L., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Proc. SEFM. pp. 3–19. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_1

[25] Beyer, D., Spiessl, M.: Reproduction package (virtual machine) for CAV 2020 article 'MetaVal: Witness validation via verification'. Zenodo (2020). https://doi.org/10.5281/zenodo.3831417

[26] Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10

[27] Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00214

[28] Beyer, D., Spiessl, M., Umbricht, S.: Reproduction package for SEFM 2022 article 'Cooperation between automatic and interactive software verifiers'. Zenodo (2022). https://doi.org/10.5281/zenodo.6541544

[29] Beyer, D., Strejček, J.: Case study on verification-witness validators: Where we are and where we go. In: Proc. SAS. pp. 160–174. LNCS 13790, Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_8

[30] Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8

[31] Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In: Proc. TACAS. pp. 1–21. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12

[32] Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Devanbu, P., Cohen, M., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 50–62. ACM (November 2020). https://doi.org/10.1145/3368089.3409718

[33] Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I. pp. 449–470. LNCS 12476, Springer (October 2020). https://doi.org/10.1007/978-3-030-61362-4_26

[34] Beyer, D., Jakobs, M.C., Lemberger, T.: Difference verification with conditions. In: d. Boer, F., Cerone, A. (eds.) Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings. pp. 133–154. LNCS 12310, Springer (September 2020). https://doi.org/10.1007/978-3-030-58768-0_8

[35] Beyer, D., Rosenfeld, M.L., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Schlingloff, B.H., Chai, M. (eds.) Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings. pp. 3–19. LNCS 13550, Springer (September 2022). https://doi.org/10.1007/978-3-031-17108-6_1

[36] Beyer, D., Spiessl, M.: Metaval: Witness validation via verification. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. pp. 165–177. LNCS 12225, Springer (July 2020). https://doi.org/10.1007/978-3-030-53291-8_10

[37] Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Schlingloff, B.H., Chai, M. (eds.) Software Engineering and Formal Methods - 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings. p. 111–128. LNCS 13550, Springer (September 2022). https://doi.org/10.1007/978-3-031-17108-6_7

[38] Black, P., Walia, K.S.: SATE VI Ockham Sound Analysis Criteria. NIST (2020)

[39] Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: FM. LNCS, vol. 8442, pp. 127–131. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_9

[40] Bühler, D.: Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C. Ph.D. thesis, University of Rennes 1, France (2017), available at https://tel.archives-ouvertes.fr/tel-01664726

[41] 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

[42] Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15

[43] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16

[44] Dardinier, T., Müller, P.: Hyper hoare logic: (dis-)proving program hyperproperties (extended version). CoRR **abs/2301.10037** (2023). https://doi.org/10.48550/arXiv.2301.10037

[45] Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: Proc. DATE. pp. 1407–1412. IEEE (2015). https://doi.org/10.7873/DATE.2015.0245

[46] Darke, P., Khanzode, M., Nair, A., Shrotri, U., Venkatesh, R.: Precise analysis of large industry code. In: Proc. APSEC. pp. 306–309. IEEE (2012). https://doi.org/10.1109/APSEC.2012.97

[47] DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)

[48] Ernst, G.: Korn: Horn clause based verification of C programs (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_36

[49] Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18

[50] Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS. pp. 69–85. LNCS 5673, Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7

[51] ISO/IEC JTC 1/SC 22: ISO/IEC 9899-2018: Information technology — Programming Languages — C. International Organization for Standardization (2018), https://www.iso.org/standard/74528.html

[52] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. NFM. pp. 41–55. LNCS 6617, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4

[53] Knight, J.C.: Safety critical systems: challenges and directions. In: Tracz, W., Young, M., Magee, J. (eds.) Proc. ICSE. pp. 547–550. ACM (2002). https://doi.org/10.1145/581339.581406

[54] Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. Formal Methods Syst. Des. **47**(1), 75–92 (2015). https://doi.org/10.1007/s10703-015-0228-1

[55] Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I., Heljanko, K.: Model checking of safety-critical software in the nuclear engineering domain. Reliab. Eng. Syst. Saf. **105**, 104–113 (2012). https://doi.org/10.1016/j.ress.2012.03.021

[56] Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: Notations and tools supporting detailed design in Java. In: OOPSLA 00: Object-Oriented Programming, Systems, Languages, and Applications. pp. 105–106. ACM (2000). https://doi.org/10.1145/367845.367996

[57] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proc. LPAR. pp. 348–370. LNCS 6355, Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20

[58] Nipkow, T., Roßkopf, S.: Isabelle's metalogic: Formalization and proof checker. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction – CADE 28. pp. 93–110. Springer International Publishing, Cham (July 2021)

[59] O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (2020). https://doi.org/10.1145/3371078

[60] Palat, J.: Introducing vagrant. Linux Journal **2012**(220), 2 (2012)

[61] Piatov, D., Janes, A., Sillitti, A., Succi, G.: Using the eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser. In: Hammouda, I., Lundell, B., Mikkonen, T., Scacchi, W. (eds.) Proc. OSS. IFIP Advances in Information and Communication Technology, vol. 378, p. 399. Springer (2012). https://doi.org/10.1007/978-3-642-33442-9_45

[62] Sagan, C.: The Demon-haunted World: Science as a Candle in the Dark. Random House (1995)

[63] Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., T-Winterhalter: Coq coq correct! Verification of type checking and erasure for coq, in coq. Proc. ACM Program. Lang. **4**(POPL), 8:1–8:28 (2020). https://doi.org/10.1145/3371076

[64] Stallman, R., McGrath, R., Smith, P.D.: GNU make. ASCII Corporation (2001)

[65] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337

[66] Wang, Q., Cao, Q.: VST-A: A foundationally sound annotation verifier. arXiv/CoRR **1909**(00097) (August 2019). https://doi.org/10.48550/arXiv.1909.00097

[67] Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) Proc. TPHOL. pp. 33–38. LNCS 5170, Springer (2008). https://doi.org/10.1007/978-3-540-71067-7_7

[68] Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. Proc. ACM Program. Lang. **7**(OOPSLA1), 522–550 (2023). https://doi.org/10.1145/3586045

A. Manuscripts

This appendix presents the complete manuscripts of the publications referenced in Chapter 4, arranged in the same chronological order as covered there. For clarity and reference, the original page numbers from each publication are retained. Consequently, readers will notice two sets of page numbers: one for this dissertation and another denoting the original pages of the reprinted article.

MetaVal: Witness Validation via Verification

Dirk Beyer and Martin Spiessl

LMU Munich, Munich, Germany

Abstract. Witness validation is an important technique to increase trust in verification results, by making descriptions of error paths (violation witnesses) and important parts of the correctness proof (correctness witnesses) available in an exchangeable format. This way, the verification result can be validated independently from the verification in a second step. The problem is that there are unfortunately not many tools available for witness-based validation of verification results. We contribute to closing this gap with the approach of *validation via verification*, which is a way to automatically construct a set of validators from a set of existing verification engines. The idea is to take as input a specification, a program, and a verification witness, and produce a new specification and a transformed version of the original program such that the transformed program satisfies the new specification if the witness is useful to confirm the result of the verification. Then, an 'off-the-shelf' verifier can be used to validate the previously computed result (as witnessed by the verification witness) via an ordinary verification task. We have implemented our approach in the validator MetaVal, and it was successfully used in SV-COMP 2020 and confirmed 3 653 violation witnesses and 16 376 correctness witnesses. The results show that MetaVal improves the effectiveness (167 uniquely confirmed violation witnesses and 833 uniquely confirmed correctness witnesses) of the overall validation process, on a large benchmark set. All components and experimental data are publicly available.

Keywords: Computer-aided verification · Software verification · Program analysis · Software model checking · Certification · Verification witnesses · Validation of verification results · Reducer

1 Introduction

Formal software verification becomes more and more important in the development process for software systems of all types. There are many verification tools available to perform verification [4]. One of the open problems that was addressed only recently is the topic of results validation [10–12,37]: The verification work is often done by untrusted verification engines, on untrusted computing infrastructure, or even on approximating computation systems, and static-analysis tools suffer from false positives that engineers in practice hate because they are tedious to refute [20]. Therefore, it is necessary to validate verification results,

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) – 378803395.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 165–177, 2020.
https://doi.org/10.1007/978-3-030-53291-8_10

ideally by an independent verification engine that likely does not have the same weaknesses as the original verifier. Witnesses also help serving as an interface to the verification engine, in order to overcome integration problems [1].

The idea to witness the correctness of a program by annotating it with assertions is as old as programming [38], and from the beginning of model checking it was felt necessary to witness counterexamples [21]. Certifying algorithms [30] are not only computing a solution but also produce a witness that can be used by a computationally much less expensive checker to (re-)establish the correctness of the solution. In software verification, witnesses became standardized[1] and exchangeable about five years ago [10,11]. In the meanwhile, the exchangeable witnesses can be used also for deriving tests from witnesses [12], such that an engineer can study an error report additionally with a debugger. The ultimate goal of this direction of research is to obtain witnesses that are certificates and can be checked by a fully trusted validator based on trusted theorem provers, such as Coq and Isabelle, as done already for computational models that are 'easier' than C programs [40].

Yet, although considered very useful, there are not many witness validators available. For example, the most recent competition on software verification (SV-COMP 2020)[2] showcases 28 software verifiers but only 6 witness validators. Two were published in 2015 [11], two more in 2018 [12], the fifth in 2020 [37], and the sixth is MetaVal, which we describe here. Witness validation is an interesting problem to work on, and there is a large, yet unexplored field of opportunities. It involves many different techniques from program analysis and model checking. However, it seems that this also requires a lot of engineering effort.

Our solution *validation via verification* is a construction that takes as input an off-the-shelf software verifier and a new program transformer, and composes a witness validator in the following way (see Fig. 1): First, the transformer takes the original input program and transforms it into a new program. In case of a violation witness, which describes a path through the program to a specific program location, we transform the program such that all parts that are marked as unnecessary for the path by the witness are pruned. This is similar to the reducer for a condition in reducer-based conditional model checking [14]. In case of a correctness witness, which describes invariants that can be used in a correctness proof, we transform the program such that the invariants are asserted (to check that they really hold) and assumed (to use them in a re-constructed correctness proof). A standard verification engine is then asked to verify that (1) the transformed program contains a feasible path that violates the original specification (violation witness) or (2) the transformed program satisfies the original specification and all assertions added to the program hold (correctness witness).

MetaVal is an implementation of this concept. It performs the transformation according to the witness type and specification, and can be configured to use any of the available software verifiers[3] as verification backend.

[1] Latest version of standardized witness format: https://github.com/sosy-lab/sv-witnesses
[2] https://sv-comp.sosy-lab.org/2020/systems.php
[3] https://gitlab.com/sosy-lab/sv-comp/archives-2020/tree/master/2020

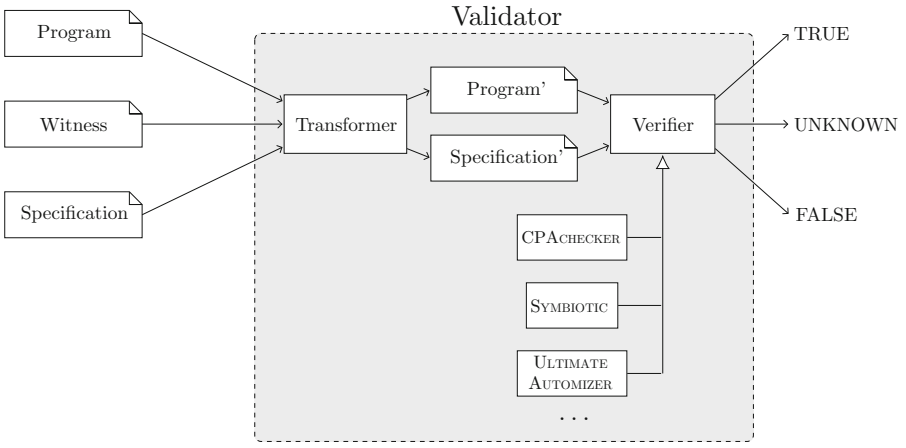


Fig. 1. Validator construction using readily available verifiers

Contributions. MetaVal contributes several important benefits:

- The program transformer was a one-time effort and is available from now on.
- Any existing standard verifier can be used as verification backend.
- Once a new verification technology becomes available in a verification tool, it can immediately be turned into a validator using our new construction.
- Technology bias can be avoided by complementing the verifier by a validator that is based on a different technology.
- Selecting the strongest verifiers (e.g., by looking at competition results) can lead to strong validators.
- All data and software that we describe are publicly available (see Sect. 6).

2 Preliminaries

For the theoretical part, we will have to set a common ground for the concepts of verification witnesses [10,11] as well as reducers [14]. In both cases, programs are represented as control-flow automata (CFAs). A *control-flow automaton* $C = (L, l_0, G)$ consists of a set L of control locations, an initial location $l_0 \in L$, and a set $G \subseteq L \times Ops \times L$ of control-flow edges that are labeled with the operations in the program. In the mentioned literature on witnesses and reducers, a simple programming language is used in which operations are either assignments or assumptions over integer variables. Operations $op \in Ops$ in such a language can be represented by formulas in first order logic over the sets V,V' of program variables before and after the transition, which we denote by $op(V, V')$. In order to simplify our construction later on, we will also allow mixed operations of the form $f(V) \wedge (x' = g(V))$ that combine assumptions with an assignment, which would otherwise be represented as an assumption followed by an assignment operation.

168 D. Beyer and M. Spiessl

```
1   void fun(uint x, uint y, uint z) {
2     if (x > y) {
3       z = 2*x-y;
4     } else {
5       z = 2*y-x+1;
6     }
7     if (z>y || z>x) {
8       return;
9     } else {
10      error();
11    }
12  }
```

Fig. 2. Example program for both correctness and violation witness validation

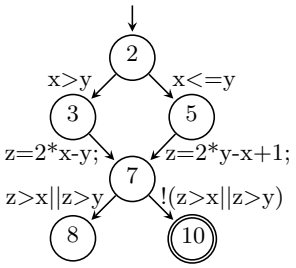


Fig. 3. CFA C of example program from Fig. 2

The conversion from the source code into a CFA and vice versa is straight forward, provided that the CFA is deterministic. A CFA is called *deterministic* if in case there are multiple outgoing CFA edges from a location l, the assumptions in those edges are mutually exclusive (but not necessarily exhaustive).

Since our goal is to validate (i.e., prove or falsify) the statement that a program fulfills a certain specification, we need to additionally model the property to be verified. For properties that can be translated into non-reachability, this can be done by defining a set $T \subseteq L$ of target locations that shall not be reached. For the example program in Fig. 2 we want to verify that the call in line 10 is not reachable. In the corresponding CFA in Fig. 3 this is represented by the reachability of the location labeled with 10. Depending on whether or not a verifier accounts for the overflow in this example program, it will either consider the program safe or unsafe, which makes it a perfect example that can be used to illustrate both correctness and violation witnesses.

In order to reason about the soundness of our approach, we need to also formalize the program semantics. This is done using the concept of concrete data states. A *concrete data state* is a mapping from the set V of program variables to their domain $\mathbb{Z}$, and a *concrete state* is a pair of control location and concrete data state. A *concrete program path* is then defined as a sequence $\pi = (c_0, l_0) \xrightarrow{g_1} \dots \xrightarrow{g_n} (c_n, l_n)$ where c_0 is the initial concrete data state, $g_i = (l_{i-1}, op_i, l_i) \in G$, and $c_{i-1}(V), c_i(V') \vDash op_i$. A *concrete execution* $ex(\pi)$ is then derived from a path π by only looking at the sequence $(c_0, l_0) \dots (c_n, l_n)$ of concrete states from the path. Note the we deviate here from the definition given in [14], where concrete executions do not contain information about the program locations. This is necessary here since we want to reason about the concrete executions that fulfill a given non-reachability specification, i.e., that never reach certain locations in the original program.

Witnesses are formalized using the concept of protocol automata [11]. A *protocol automaton* $W = (Q, \Sigma, \delta, q_0, F)$ consists of a set Q of states, a set of transition labels $\Sigma = 2^G \times \Phi$, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, an initial state q_0, and a set $F \subseteq Q$ of final states. A state is a pair that consists of a name to identify

the state and a predicate over the program variables V to represent the state invariant.[4] A transition label is a pair that consists of a subset of control-flow edges and a predicate over the program variables V to represent the guard condition for the transition to be taken. An *observer automaton* [11,13,32,34,36] is a protocol automaton that does not restrict the state space, i.e., if for each state $q \in Q$ the disjunction of the guard conditions of all outgoing transitions is a tautology. Violation witnesses are represented by protocol automata in which all state invariants are *true*. Correctness witnesses are represented by observer automata in which the set of final states is empty.

3 Approach

3.1 From Witnesses to Programs

When given a CFA $C = (L, l_0, G)$, a specification $T \subseteq L$, and a witness automaton $W = (Q, \Sigma, \delta, q_0, F)$, we can construct a product automaton $A_{C \times W} = (L \times Q, (l_0, q_0), \Gamma, T \times F)$ where $\Gamma \subseteq (L \times Q) \times (Ops \times \Phi) \times (L \times Q)$. The new transition relation Γ is defined by allowing for each transition g in the CFA only those transitions (S, φ) from the witness where $g \in S$ holds:

$$\Gamma = \left\{ \left((l_i, q_i), (op, \varphi), (l_j, q_j) \right) \mid \exists S : \left(q_i, (S, \varphi), q_j \right) \in \delta, (l_i, op, l_j) \in S \right\}$$

We can now define the semantics of a witness by looking at the paths in the product automaton and mapping them to concrete executions in the original program. A path of the product automaton $A_{C,W}$ is a sequence $(l_0, q_0) \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{n-1}} (l_n, q_n)$ such that $\left((l_i, q_i), \alpha_i, (l_{i+1}, q_{i+1}) \right) \in \Gamma$ and $\alpha_i = (op_i, \phi_i)$.

It is evident that the automaton $A_{C \times W}$ can easily be mapped to a new program $C_{C \times W}$ by reducing the pair (op, φ) in its transition relation to an operation $\overline{op}$. In case op is a pure assumption of the form $f(V)$ then $\overline{op}$ will simply be $f(V) \wedge \varphi(V)$. If op is an assignment of the form $f(V) \wedge (x' = g(V))$, then $\overline{op}$ will be $(f(V) \wedge \varphi(V)) \wedge (x' = g(V))$. This construction has the drawback that the resulting CFA might be non-deterministic, but this is actually not a problem when the corresponding program is only used for verification. The non-determinism can be expressed in the source code by using non-deterministic values, which are already formalized by the community and established in the SV-COMP rules, and therefore also supported by all participating verifiers. The concrete executions of $C_{C \times W}$ can be identified with concrete executions of C by projecting their pairs (l, q) on their first element. Let $proj_C(ex(C_{C \times W}))$ denote the set of concrete executions that is derived this way. Due to how the relation Γ of $A_{C \times W}$ is constructed, it is guaranteed that this is a subset of the executions of C, i.e., $proj_C(ex(C_{C \times W})) \subseteq ex(C)$. In this respect the witness acts in very much the same way as a reducer [14], and the reduction of the search space is also one of the desired properties of a validator for violation witnesses.

[4] These invariants are the central piece of information in correctness witnesses. While invariants that proof a program correct can be hard to come up with, they are usually easier to check.

170 D. Beyer and M. Spiessl

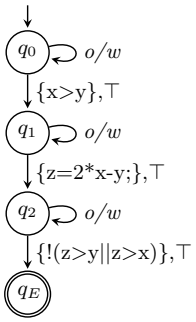


Fig. 4. Violation witness W_V

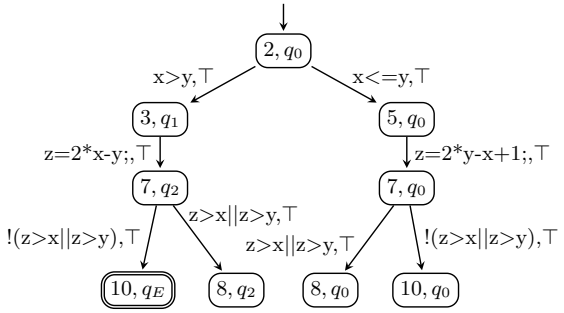


Fig. 5. Product automaton $A_{C\times W_V}$

3.2 Programs from Violation Witnesses

For explaining the validation of results based on a violation witness, we consider the witness in Fig. 4 for our example C program in Fig. 2. The program $C_{C\times W_V}$ resulting from product automaton $A_{C\times W_V}$ in Fig. 5 can be passed to a verifier. If this verification finds an execution that reaches a specification violation, then this violation is guaranteed to be also present in the original program. There is however one caveat: In the example in Fig. 5, a reachable state $(10, q_0)$ at program location 10 (i.e., a state that violates the specification) can be found that is not marked as accepting state in the witness automaton W_V. For a strict version of witness validation, we can remove all states that are in $T \times Q$ but not in $T \times F$ from the product automaton, and thus, from the generated program. This will ensure that if the verifier finds a violation in the generated program, the witness automaton also accepts the found error path. The version of MetaVal that was used in SV-COMP 2020 did not yet support strict witness validation.

3.3 Programs from Correctness Witnesses

Correctness witnesses are represented by observer automata. Figure 6 shows a potential correctness witness W_C for our example program C in Fig. 2, where the invariants are annotated in bold font next to the corresponding state. The construction of the product automaton $A_{C\times W_C}$ in Fig. 7 is a first step towards reestablishing the proof of correctness: the product states tell us to which control locations of the CFA for the program the invariants from the witness belong.

The idea of a result validator for correctness witnesses is to

1. check the invariants in the witness and
2. use the invariants to establish that the original specification holds.

We can achieve the second goal by extracting the invariants from each state in the product automaton $A_{C\times W_C}$ and adding them as conditions to all edges by which the state can be reached. This will then be semantically equivalent to assuming that the invariants hold at the state and potentially make the consecutive proof easier. For soundness we need to also ensure the first goal. To achieve that, we add transitions into a (new) accepting state from $T \times F$ whenever we transition

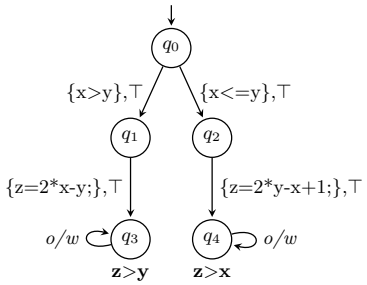


Fig. 6. Correctness witness W_C

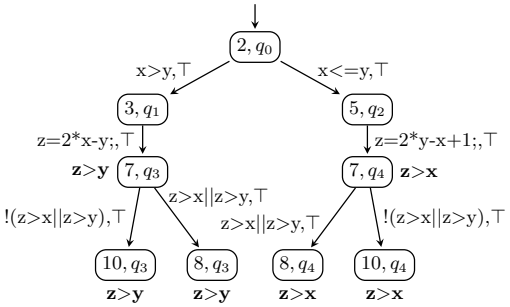


Fig. 7. Product automaton $A_{C \times W_C}$

into a state q and the invariant of q does not hold, and we add self-loops such that the automaton stays in the new accepting state forever. In sum, for each invariant, there are two transitions, one with the invariant as guard (to assume that the invariant holds) and one with the negation of the invariant as guard (to assert that the invariant holds, going to an accepting (error) state if it does not hold). This transformation ensures that the resulting automaton after the transformation is still a proper observer automaton.

4 Evaluation

This section describes the results that were obtained in the 9th Competition on Software Verification (SV-COMP 2020), in which MetaVal participated as validator. We did not perform a separate evaluation because the results of SV-COMP are complete, accurate, and reproducible; all data and tools are publicly available for inspection and replication studies (see data availability in Sect. 6).

4.1 Experimental Setup

Execution Environment. In SV-COMP 2020, the validators were executed in a benchmark environment that makes use of a cluster with 168 machines, each of them having an Intel Xeon E3-1230 v5 CPU with 8 processing units, 33 GB of RAM, and the GNU/Linux operating system Ubuntu 18.04. Each validation run was limited to 2 processing units and 7 GB of RAM, in order to allow up to 4 validation runs to be executed on the same machine at the same time. The time limit for a validation run was set to 15 min for correctness witnesses and to 90 s for violation witnesses. The benchmarking framework BenchExec 2.5.1 was used to ensure that the different runs do not influence each other and that the resource limits are measured and enforced reliably [15]. The exact information to replicate the runs of SV-COMP 2020 can be found in Sect. 3 of the competition report [4].

Benchmark Tasks. The verification tasks[5] of SV-COMP can be partitioned wrt. their specification into ReachSafety, MemSafety, NoOverflows, and Termination. Validators can be configured using different options for each specification.

[5] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20

Table 1. Overview of validation for violation witnesses in SV-COMP 2020

Specification	Measure	CPAchecker	CPA-wtt	FShell-wtt	MetaVal	NITWIT	UAutomizer
ReachSafety (35 652 witnesses)	executed on	35 652	25 812	25 812	35 652	21 636	25 812
	uniquely confirmed	3 043	42	175	44	398	547
	jointly confirmed	8 019	6 010	6 740	1 566	8 055	3 802
Termination (9 720 witnesses)	executed on	3 043			9 720		9 720
	uniquely confirmed	566			9		235
	jointly confirmed	1 539			256		1 493
NoOverflow (3 149 witnesses)	executed on	3 149	3 149	3 149	3 149		3 149
	uniquely confirmed	6	1	31	1		89
	jointly confirmed	1 668	1 067	1 267	1 186		1 590
MemSafety (2 681 witnesses)	executed on	2 681	2 213	2 681	2 681		2 681
	uniquely confirmed	278	0	21	113		44
	jointly confirmed	737	250	364	478		372

Table 2. Overview of validation for correctness witnesses in SV-COMP 2020

Specification	Measure	CPAchecker	MetaVal	UAutomizer
ReachSafety (66 435 witnesses)	executed on	66 435	66 435	66 435
	uniquely confirmed	1 750	391	708
	jointly confirmed	17 592	13 862	16 834
NoOverflow (3 179 witnesses)	executed on		3 179	3 179
	uniquely confirmed		44	74
	jointly confirmed		870	870
MemSafety (4 426 witnesses)	executed on		4 426	4 426
	uniquely confirmed		398	173
	jointly confirmed		811	811

Validator Configuration. Since our architecture (cf. Fig. 1) allows for a wide range of verifiers to be used for validation, there are many interesting configurations for constructing a validator. Exploring all of these in order to find the best configuration, however, would require significant computational resources, and also be susceptible to over-fitting. Instead, we chose a heuristic based on the results of the competition from the previous year, i.e., SV-COMP 2019 [3]. The idea is that a verifier which performed well at *verifying* tasks for a specific specification is also a promising candidate to be used in *validating* results for that specification. Therefore the configuration of our validator MetaVal uses CPA-Seq as verifier for tasks with specification ReachSafety, Ultimate Automizer for NoOverflow and Termination, and Symbiotic for MemSafety.

4.2 Results

The results of the validation phase in SV-COMP 2020 [5] are summarized in Table 1 (for violation witnesses) and Table 2 (for correctness witnesses). For each specification, MetaVal was able to not only confirm a large number of results

that were also validated by other tools, but also to confirm results that were not previously validated by any of the other tools.[6]

For violation witnesses, we can observe that MetaVal confirms significantly less witnesses than the other validators. This can be explained partially by the restrictive time limit of 90 s. Our approach not only adds overhead when generating the program from the witness, but this new program can also be harder to parse and analyze for the verifier we use in the backend. It is also the case that the verifiers that we use in MetaVal are not tuned for such a short time limit, as a verifier in the competition will always get the full 15 min. For specification ReachSafety, for example, we use CPA-Seq, which starts with a very simply analysis and switches verification strategies after a fixed time that happens to be also 90 s. So in this case we will never benefit from the more sophisticated strategies that CPA-Seq offers.

For validation of correctness witnesses, where the time limit is higher, this effect is less noticeable such that the number of results confirmed by MetaVal is more in line with the numbers achieved by the other validators. For specification MemSafety, MetaVal even confirms more correctness witnesses than Ultimate Automizer. This indicates that Symbiotic was a good choice in our configuration for that specification. Symbiotic generally performs much better in verification of MemSafety tasks than Ultimate Automizer, so this result was expected.

Before the introduction of MetaVal, there was only one validator for correctness witnesses in the categories NoOverflow and MemSafety, while constructing a validator for those categories with our approach did not require any additional development effort.

5 Related Work

Programs from Proofs. Our approach for generating programs can be seen as a variant of the Programs from Proofs (PfP) framework [27,41]. Both generate programs from an abstract reachability graph of the original program. The difference is that PfP tries to remove all specification violations from the graph, while we just encode them into the generated program as violation of the standard reachability property. We do this for the original specification and the invariants in the witness, which we treat as additional specifications.

Automata-Based Software Model Checking. Our approach is also similar to that of the validator Ultimate Automizer [10]. For violation witnesses, it also constructs the product of CFA and witness. For correctness witnesses, it instruments the invariants directly into the CFA of the program (see [10], Sect. 4.2) and passes the result to its verification engine, while MetaVal constructs the product of CFA and witness, and applies a similar instrumentation. In both cases, MetaVal's transformer produces a C program, which can be passed to an independent verifier.

Reducer-Based Conditional Model Checking. The concept of generating programs from an ARG has also been used to successfully construct conditional verifiers [14].

[6] In the statistics, a witness is only counted as confirmed if the verifier correctly stated whether the input program satisfies the respective specification.

Our approach for correctness witnesses can be seen as a special case of this technique, where MetaVal acts as initial verifier that does not try to reduce the search space and instead just instruments the invariants from the correctness witness as additional specification into the program.

Verification Artifacts and Interfacing. The problem that verification results are not treated well enough by the developers of verification tools is known [1] and there are also other works that address the same problem, for example, the work on execution reports [19] or on cooperative verification [17].

Test-Case Generation. The idea to generate test cases from verification counterexamples is more than ten years old [8,39], has since been used to create debuggable executables [31,33], and was extended and combined to various successful automatic test-case generation approaches [24,25,29,35].

Execution. Other approaches [18,22,28] focus on creating tests from concrete and tool-specific counterexamples. In contrast, witness validation does not require full counterexamples, but works on more flexible, possibly abstract, violation witnesses from a wide range of verification tools.

Debugging and Visualization. Besides executing a test, it is important to understand the cause of the error path [23], and there are tools and methods to debug and visualize program paths [2,9,26].

6 Conclusion

We address the problem of constructing a tool for witness validation in a systematic and generic way: We developed the concept of *validation via verification*, which is a two-step approach that first applies a program transformation and then applies an off-the-shelf verification tool, without development effort.

The concept is implemented in the witness validator MetaVal, which has already been successfully used in SV-COMP 2020. The validation results are impressive: the new validator enriches the competition's validation capabilities by 164 uniquely confirmed violation results and 834 uniquely confirmed correctness results, based on the witnesses provided by the verifiers. This paper does not contain an own evaluation, but refers to results from the recent competition in the field.

The major benefit of our concept is that it is now possible to configure a spectrum of validators with different strengths, based on different verification engines. The 'time to market' of new verification technology into validators is negligibly small because there is no development effort necessary to construct new validators from new verifiers. A potential technology bias is also reduced.

Data Availability Statement. All data from SV-COMP 2020 are publicly available: witnesses [7], verification and validation results as well as log files [5], and benchmark programs and specifications [6][7]. The validation statistics in Tables 1 and 2 are available in the archive [5] and on the SV-COMP website[8]. MetaVal 1.0 is available on GitLab[9] and in our AEC-approved virtual machine [16].

[7] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20
[8] https://sv-comp.sosy-lab.org/2020/results/results-verified/validatorStatistics.html
[9] https://gitlab.com/sosy-lab/software/metaval/-/tree/1.0

References

1. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making software verification tools really work. In: Proc. ATVA, LNCS, vol. 6996, pp. 28–42. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_3
2. Artho, C., Havelund, K., Honiden, S.: Visualization of concurrent program executions. In: Proc. COMPSAC, pp. 541–546. IEEE (2007). https://doi.org/10.1109/COMPSAC.2007.236
3. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3), LNCS, vol. 11429, pp. 133–155. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_9
4. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2), LNCS, vol. 12079, pp. 347–367. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_21
5. Beyer, D.: Results of the 9th International Competition on Software Verification (SV-COMP 2020). Zenodo (2020). https://doi.org/10.5281/zenodo.3630205
6. Beyer, D.: SV-Benchmarks: Benchmark set of 9th Intl. Competition on Software Verification (SV-COMP 2020). Zenodo (2020). https://doi.org/10.5281/zenodo.3633334
7. Beyer, D.: Verification witnesses from SV-COMP 2020 verification tools. Zenodo (2020). https://doi.org/10.5281/zenodo.3630188
8. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE, pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
9. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Proc. CAV (2), LNCS, vol. 9780, pp. 502–509. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_28
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE, pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE, pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
12. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP, LNCS, vol. 10889, pp. 3–23. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92994-1_1
13. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_16
14. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE, pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
15. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
16. Beyer, D., Spiessl, M.: Replication package (virtual machine) for article 'MetaVal: Witness validation via verification' in Proc. CAV 2020. Zenodo (2020). https://doi.org/10.5281/zenodo.3831417
17. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. arXiv/CoRR **1905**(08505), May 2019. https://arxiv.org/abs/1905.08505

18. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. In: Proc. CCS, pp. 322–335. ACM (2006). https://doi.org/10.1145/1180405.1180445
19. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Proc. ASE, pp. 200–205. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115633
20. Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: Proc. ASE, pp. 332–343. ACM (2016). https://doi.org/10.1145/2970276.2970347
21. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proc. DAC, pp. 427–432. ACM (1995). https://doi.org/10.1145/217474.217565
22. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proc. ICSE, pp. 422–431. ACM (2005). https://doi.org/10.1145/1062455.1062533
23. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Proc. FM, LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_17
24. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. PLDI, pp. 213–223. ACM (2005). https://doi.org/10.1145/1065010.1065036
25. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: A new algorithm for property checking. In: Proc. FSE, pp. 117–127. ACM (2006). https://doi.org/10.1145/1181775.1181790
26. Gunter, E.L., Peled, D.A.: Path exploration tool. In: Proc. TACAS, LNCS, vol. 1579, pp. 405–419. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_28
27. Jakobs, M.C., Wehrheim, H.: Programs from proofs: A framework for the safe execution of untrusted software. ACM Trans. Program. Lang. Syst. **39**(2), 7:1–7:56 (2017). https://doi.org/10.1145/3014427
28. Li, K., Reichenbach, C., Csallner, C., Smaragdakis, Y.: Residual investigation: Predictive and precise bug detection. In: Proc. ISSTA, pp. 298–308. ACM (2012). https://doi.org/10.1145/2338965.2336789
29. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proc. ICSE, pp. 416–426. IEEE (2007). https://doi.org/10.1109/ICSE.2007.41
30. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009
31. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Proc. FM, LNCS, vol. 6664, pp. 73–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_8
32. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. **28**(11), 1056–1076 (2002). https://doi.org/10.1109/TSE.2002.1049404
33. Rocha, H., Barreto, R.S., Cordeiro, L.C., Neto, A.D.: Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: Proc. IFM, LNCS, vol. 7321, pp. 128–142. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30729-4_10
34. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000). https://doi.org/10.1145/353323.353382

35. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proc. FSE, pp. 263–272. ACM (2005). https://doi.org/10.1145/1081706.1081750
36. Šerý, O.: Enhanced property specification and verification in BLAST. In: Proc. FASE, LNCS, vol. 5503, pp. 456–469. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_32
37. Svejda, J., Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NITWIT. In: Proc. TACAS, LNCS, vol. 12078, pp. 40–57. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_3
38. Turing, A.: Checking a large routine. In: Report on a Conference on High Speed Automatic Calculating Machines, pp. 67–69. Cambridge Univ. Math. Lab. (1949)
39. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test-input generation with Java PATHFINDER. In: Proc. ISSTA, pp. 97–107. ACM (2004). https://doi.org/10.1145/1007512.1007526
40. Wimmer, S., von Mutius, J.: Verified certification of reachability checking for timed automata. In: Proc. TACAS, LNCS, vol. 12078, pp. 425–443. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_24
41. Wonisch, D., Schremmer, A., Wehrheim, H.: Programs from proofs: A PCC alternative. In: Proc. CAV, LNCS, vol. 8044, pp. 912–927. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_65

The Static Analyzer Frama-C in SV-COMP (Competition Contribution)

Dirk Beyer ✉ and Martin Spiessl

LMU Munich, Munich, Germany

Abstract. Frama-C is a well-known platform for source-code analysis of programs written in C. It can be extended via its plug-in architecture by various analysis backends and features an extensive annotation language called ACSL. So far it was hard to compare Frama-C to other software verifiers. Our competition participation contributes an adapter named Frama-C-SV, which makes it possible to evaluate Frama-C against other software verifiers. The adapter transforms standard verification tasks (from the well-known SV-Benchmarks collection) in a way that can be understood by Frama-C and produces a verification witness as output. While Frama-C provides many different analyses, we focus on the Evolved Value Analysis (EVA), which uses a combination of different domains to over-approximate the behavior of the analyzed program.

Keywords: Software verification · Program analysis · Formal methods · Competition on Software Verification · Comparative Evaluation · SV-COMP · Frama-C

1 Approach

This competition contribution is based on Frama-C [12], a program-analysis platform for C programs. The purpose of the participation in the comparative evaluation SV-COMP is to show the strengths of Frama-C when applied to the problem of verifying C programs from the SV-Benchmarks [4] collection of verification tasks.

2 Architecture

Although Frama-C has a large configuration space, it does not support standard specifications as used in SV-COMP, and it does not produce verification witnesses as default. In order to overcome this obstacle we implemented an adapter for Frama-C using input and output transformers, and the adaption architecture is illustrated in Fig. 1. In the following, we describe the artifacts and actors of the participating verifier: in Sect. 2.1 we describe all the components that are developed as part of the adapter, while in Sect. 2.2 we describe in more detail how the used EVA analysis of Frama-C works.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 429–434, 2022.
https://doi.org/10.1007/978-3-030-99527-0_26

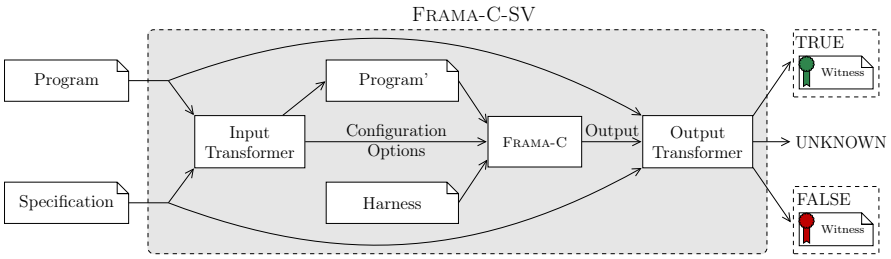


Fig. 1: Architecture of Frama-C-SV: the inputs and outputs of Frama-C are translated to interface with the established standards as used by SV-COMP; the components that are necessary to adapt Frama-C for comparison with other verifiers amount to 678 lines of code mostly written in Python

2.1 Frama-C-SV

Input Transformer. The input transformer takes the program p and specification s and creates a new program p' in which the specification s has been expressed as Frama-C-specific annotations. Frama-C uses ACSL [1] as language to specify annotations. The input transformer also selects configuration parameters for Frama-C that are best suited for the verification task. Currently we encode reachability tasks into signed integer overflows by adding an artificial overflow to the body of the function `reach_error`. This works well in practice and is also sound, since if there were any other overflows, the task would contain undefined behavior and would not be a valid reachability task in the first place.

Configuration Options. Depending on the input program and specification, we can choose different options that are passed to Frama-C. In essence, this acts like an algorithm selection [14] and, e.g., allows us to choose a different configuration of Frama-C depending on the specified property.

Harness. Some programs in the SV-Benchmarks collection use specific functions to model non-determinism. We provide implementations for those functions (`__VERIFIER_*`) in a separate C program such that the semantics of those functions can be understood by Frama-C. This separate C program is passed to Frama-C together with the transformed program p'.

Output Transformer. The output of Frama-C needs to be interpreted regarding the original specification, and depending on the outcome, a verification witness needs to be generated. Thus, we need an output transformer for (a) providing a verdict for the verification task and (b) providing a verification witness. Regarding (a), the output transformer interprets the CSV report that can be generated by Frama-C to determine whether the program was proven to be safe (verdict `TRUE`), whether a specification violation occurred (verdict `FALSE`), or whether no such statement can be made (verdict `UNKNOWN`). We also generate a minimal correctness or violation witness for the verdicts `TRUE` and `FALSE`, respectively.

The witness automata consist of only one node, which for violation witnesses is marked as violation node. In the future we plan to augment these witnesses with information such as invariants that have been found by Frama-C.

2.2 Frama-C

One of the strengths of Frama-C is its modular architecture [10], which allows a configuration of the best possible analysis backends for a certain verification problem. We choose the plug-in EVA [9], which is well suited for an automatic analysis. Other plug-ins such as the Weakest-Preconditions (WP) plug-in require hints from the user in order to be effective. In the following we will briefly describe the most important aspects of the EVA analysis configuration that we use. For a more detailed description, we refer the reader to the relevant literature [7, 8, 9].

Frama-C provides a meta-option called `-eva-precision` for the EVA plug-in with possible values ranging from 0 to 11. With higher values for this option more precise domains and thresholds are used, at the cost of increased computation time. We currently use the maximum value of 11 in order to make the best use of the 900 s CPU time limit. In the future we might want to iteratively increase this value starting at lower precisions.

Domains. The EVA analysis always uses the domain *cvalue*, which tracks values of variables either as constant values, sets, or intervals of possible values (including modular congruence constraints). For pointer addresses, these are either tracked as addresses with offsets or as so-called garbled mix, which overapproximates the set of possible memory locations. In addition, depending on the precision level, various other domains are used that we describe in the following. The domain *symbolic-locations* tracks a map of symbolic locations to values, which is, e.g., helpful for analyzing expressions containing array accesses such as `a[i]<a[j]`. The *equality* domain tracks equalities of C expressions found in the code, whereas the *gauges* domain tracks relations between variables in a loop with the goal to discover linear inequality invariants [16]. Lastly the *octagon* domain tracks certain linear constraints between pairs of variables [13]. As we use the highest precision level, all of these domains are used in our contribution.

Precision of the State-Space Exploration. Apart from the domains, the precision of state-space exploration in Frama-C is affected by various options. We will describe some of these in the following; a complete list of affected settings and values is always printed by Frama-C when the option *eva-precision* is specified by the user. Option *slevel* (set to 5 000) determines how many separate states are kept before new states will be joined into existing ones. Option *ilevel* (set to 256) determines how many different values are tracked per variable before overapproximating the value range. Option *plevel* (set to 2 000) affects the size up to which arrays are tracked. The option *auto-loop-unroll* (set to 1 024) will determine up to which bound a loop is considered for unrolling.

3 Strengths and Weaknesses

The competition contribution shows the strengths of Frama-C in checking C programs for overflows and also —in the currently supported sub-categories [1]— for reachability. Here we are able to show that our results are comparable and often surpass those of other tools based on abstract interpretation [11] such as Goblint [15]. While the EVA analysis of Frama-C that we use is based on abstract interpretation, the precision options described in Sect. 2.2 allow for a more precise state-space exploration, which behaves more like model checking. More details about the results can be found in the competition report [2] and artifact [3].

The approach that we describe in this paper creates a compatibility layer between the abilities used by Frama-C and the standards used in the SV-Benchmarks collection. While still a work in progress, we have shown that it is possible to bridge this gap while preserving overall soundness. It is also interesting to consider the results on verification tasks from the SV-Benchmarks collections for a tool that did not participate before.

Although our approach is sound in general, we are likely not showcasing the full potential of Frama-C. One aspect to consider here is the large configuration space, which means there might be ways to verify more tasks with a better heuristic for selecting the configuration options. The other aspect is that Frama-C also provides different plug-ins such as the WP plug-in, which requires more (manual) annotations, but can also potentially solve more tasks than the more automatic EVA plug-in.

4 Software Project and Contributors

The software project Frama-C is developed at `https://git.frama-c.com/pub/frama-c/` and our adapter Frama-C-SV is developed at `https://gitlab.com/sosy-lab/software/frama-c-sv`, both being released under open-source licenses. The exact version of the adapter that participated in SV-COMP 2022 is also archived in the competition's tool-archive repository [2] [6]. Frama-C was funded by the European Commission in program Horizon 2020. The adapter Frama-C-SV was funded by the DFG. We thank the Frama-C authors [3] for their contribution to the software-verification community.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [2] and available on the competition web site. This includes the verification tasks [4], competition results [3], verification witnesses [5], scripts, and instructions for reproduction. The version of Frama-C-SV as used in the competition is archived together with other participating tools [6].

Funding Statement. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

[1] We opted out of subcategories with unsound results caused by Frama-C making assumptions that are different from the conventions of SV-COMP.

[2] `https://gitlab.com/sosy-lab/sv-comp/archives-2022/blob/svcomp22/2022/frama-c-sv.zip`

[3] `https://frama-c.com/html/authors.html`

References

1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
3. Beyer, D.: Results of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831008
4. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
5. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5838498
6. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
7. Blazy, S., Bühler, D., Yakobowski, B.: Structuring abstract interpreters through state and value abstractions. In: Proc. VMCAI. pp. 112–130. LNCS 10145, Springer (2017). https://doi.org/10.1007/978-3-319-52234-0_7
8. Bühler, D.: Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C. Ph.D. thesis, University of Rennes 1, France (2017), available at https://tel.archives-ouvertes.fr/tel-01664726
9. Bühler, D., Cuoq, P., Yakobowski, B., Lemerre, M., Maroneze, A., Perelle, V., Prevosto, V.: Eva: The Evolved Value Analysis plug-in (2020), available at https://frama-c.com/download/frama-c-eva-manual.pdf
10. Correnson, L., Cuoq, P., Kirchner, F., Maroneze, A., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C user manual (2020), available at https://frama-c.com/download/frama-c-user-manual.pdf
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL. pp. 238–252. ACM (1977)
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
13. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
14. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
15. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints (competition contribution). In: Proc. TACAS (2). pp. 438–442. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_28
16. Venet, A.: The gauge domain: Scalable analysis of linear inequality invariants. In: Proc. CAV. pp. 139–154. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_15

Cooperation Between Automatic and Interactive Software Verifiers

Dirk Beyer, Martin Spiessl, and Sven Umbricht

LMU Munich, Munich, Germany

Abstract. The verification community develops two kinds of verification tools: automatic verifiers and interactive verifiers. There are many such verifiers available, and there is steady progress in research. However, cooperation between the two kinds of verifiers was not yet addressed in a modular way. Yet, it is imperative for the community to leverage all possibilities, because our society heavily depends on software systems that work correctly. This paper contributes tools and a modular design to address the open problem of insufficient support for cooperation between verification tools. We identify invariants as information that needs to be exchanged in cooperation, and we support translation between two 'containers' for invariants: program annotations and correctness witnesses. Using our new building blocks, invariants computed by automatic verifiers can be given to interactive verifiers as annotations in the program, and annotations from the user or interactive verifier can be given to automatic verifiers, in order to help the approaches mutually to solve the verification problem. The modular framework, and the design choice to work with readily-available components in off-the-shelf manner, opens up many opportunities to combine new tools from existing components. Our experiments on a large set of programs show that our constructions work, that is, we constructed tool combinations that can solve verification tasks that the verifiers could not solve before.

Keywords: Software verification, Program analysis, Invariant generation, Automatic verification, Interactive verification, CPAchecker, Frama-C

1 Introduction

Software verification becomes more and more important, and large IT companies are investing into this technology [5,25,29]. There was a lot of progress in the past two decades and many software-verification tools exist [7,8,15,34,42]. But there are also obstacles that hinder the application of new technology in practice [3,35]. The verification tools can roughly be divided into two different flavors: automatic verifiers, which are more suited for automatic settings such as continuous-integration checks, and interactive verifiers, which can be fed with proof hints to solve verification tasks. These different tools have different strengths and often one verifier alone is not able to prove the correctness. Yet, the

B.-H. Schlingloff and M. Chai (Eds.): SEFM 2022, LNCS 13550, pp. 111–128, 2022.
https://doi.org/10.1007/978-3-031-17108-6_7

potential from cooperation between different kinds of verifiers is a largely unused technology, although it is expected to significantly improve the state of the art.

In this paper, we contribute ideas to bridge the gap between automatic and interactive verifiers by introducing cooperation between tools of both kinds. As a starting point, we identify invariants as the objects that we need to exchange. Then we investigate which interfaces are supported by different verification tools. As a result, we choose verification witnesses [12] and annotations [6] as containers for the invariants. We implement various transformers for exchanging invariants between the different interfaces. This results in a modular composition framework that is based on off-the-shelf components (in binary format). We can use existing components because we base our work on existing interfaces (witnesses and annotations).

Automatic verifiers, such as CBMC [28], CPACHECKER [18], GOBLINT [49], KORN [32], PESCO [48], SYMBIOTIC [26], ULTIMATE AUTOMIZER [39], and VERIABS [1] (alphabetic order, just to name a few, for a larger list we refer to a competition report [8]), usually take as input a program and a specification (a.k.a. verification task) and compute invariants, in order to prove correctness. The above-mentioned verifiers can save the computed invariants into a standard witness file for later use (e.g., for result validation).

Interactive verifiers, such as DAFNY [46], FRAMA-C [30], KEY [2], KIV [33], and VERIFAST [43] (alphabetic order, just to name a few, for a larger list we refer to a competition report [34]), usually take as input a program with an inlined specification (contracts, asserts), and during the verification process, the verification engineer can interact with the verifier by providing invariants and other information as annotations in the program.

The automatic verifiers use a standardized exchange format for verification witnesses [12], and thus, we can easily plug-in all of them. The interactive verifiers come each with their own annotation language. We decided to consider only ACSL [6], which is supported by FRAMA-C [30], as a starting point for our study, because it is well documented. In practice, many of these annotation languages are similar, so our results apply to other annotation languages as well.

Contributions. This paper contributes the following in order to enable new verification technology:

- We develop a novel compositional design to construct new tools for software verification from existing 'off-the-shelf' components:
  1. We construct interactive verifiers from automatic verifiers and validators.
  2. We construct result validators from interactive verifiers.
  3. We improve interactive verifiers by feeding them with invariants computed by automatic verifiers.
- We identified an appropriate benchmark set of verification tasks with verification witnesses that contain provably useful invariants. We also created second benchmark set with manually added ACSL annotations containing (inductive) loop invariants and assertions. In order to make our evaluation reproducible and to offer the invariants to other researchers for further experiments, we make both benchmark sets available.

- We make all components and transformations available as open source, such that other researchers and practitioners can reuse and experiment with them, and verify our results (see Sect. 5 for the data-availability statement).
- We perform a sound experimental evaluation on a large benchmark set to investigate the effectivity of the new compositions. The results are promising and suggest that such compositions are worth to be considered in practice.

Combinations like the proposed cooperation approach can significantly impact the way in which verification tools are used in practice. Currently, engineers need to use both kinds of verifiers, automatic and interactive, in isolation, but our study has shown that there is much potential in leveraging cooperation.

Related Work. In the following we discuss the most related existing approaches.

Transform Programs. This is not the first work to convert the semantics of witness validation into a program. Some existing approaches [14] focus on violation witnesses, while we solely focus on correctness witnesses. Most similar in this regard is MetaVal [21]. The main difference is that we preserve the program structure while MetaVal does an automaton product between the *control-flow automaton (CFA)* of the program and witness automaton, and turns the result back into a C program, which will result in a different syntactic structure.

Interact via Conditions. The approach *conditional model checking* [16] also achieves cooperation between verifiers, but is limited to automatic verifiers that support the condition format and the verifier that comes second uses the condition to restrict the part of the state space that is explored. Our framework supports more tools via the usage of standardized exchange formats, also considers interactive verifiers, and the second verifier still performs a full proof. Another approach that builds on conditions is *alternating conditional analysis* [36,37]. Here, the witness format is also used as standardized exchange format and multiple verifiers are supported. However, the focus is on violation witnesses whereas we are focussing on correctness witnesses. Instead of removing parts of the state space, we actually extend the property that needs to be checked, such that it is (potentially) easier to be proven. The same holds if we compare our component Witness2Assert to *reducer-based conditional model checking* [17]. While both approaches encode the important information into the original program, we actually would need to assume the invariants instead of asserting them in order to act as a reducer. Conditions are also used to improve testing [19,27,31].

Store and Exchange Proofs. Another parallel can be drawn to *proof-carrying code* [44,45,47], where the proof of correctness is stored alongside the program. We do the same here in cases where the added annotations actually suffice for a full proof by Frama-C, but we also have the possibility to generate partial proofs. Correctness witnesses are used to store intermediate results and to validate results [11]. Proofs are also stored in the area of theorem provers [38] (https://www.isa-afp.org/) and SAT solvers [40,41].

114 D. Beyer, M. Spiessl, and S. Umbricht

```
1
2  int main() {
3    unsigned int x = 0;
4    unsigned int y = 0;
5
6    while (nondet_int()) {
7      x++;
8
9      y++;
10   }
11   assert(x==y);
12   return 0;
13 }
```

Fig. 1. Example program with loop invariant `x==y`

```
1  //@ensures \return==0;
2  int main() {
3    unsigned int x = 0;
4    unsigned int y = 0;
5    //@loop invariant x==y;
6    while (nondet_int()) {
7      x++;
8      //@assert x==y+1;
9      y++;
10   }
11   assert(x==y);
12   return 0;
13 }
```

Fig. 2. Example program with ACSL annotations

2 Preliminaries

For our framework that enables cooperation between automatic and interactive verifiers we need to take into account the interfaces that each of them provide, i.e., how the information important for the verification process is communicated. For automatic verifiers there exists a common exchange format [12] in which verifiers export the program invariants they found. For interactive verifiers, we look at ACSL [6], the specification language that is e.g. used by Frama-C. In the following, we will quickly introduce these formats and the general verification problem we are looking at using a small example program that is depicted in Fig. 1.

For the rest of the paper, we will focus on reachability properties, though our approach can also be extended to work for other properties as well.[1] The crucial part of verifying reachability properties is to find the right loop invariants. In the example program this would be the fact that `x==y` always holds before each loop iteration. Please note that while this invariant is also present in the assertion in line 11, for more complicated programs it is generally not the case that we can find the invariants written in the code. Also, since there might be more than one loop in a program, a verifier might only partially succeed and therefore only be able to provide invariants for some of these loops, or only invariants that are not yet strong enough to prove the program correct. This is why cooperation by exchange of these discovered invariants can potentially lead to better results.

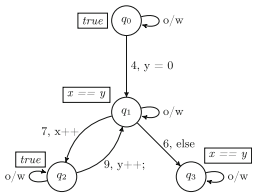
2.1 Verification Witnesses

In case an automatic verifier can prove our example program correct, information like a discovered invariant is normally made available as shown in Fig. 3a in the standard witness exchange format (described in [12], maintained at https://github.com/sosy-lab/sv-witnesses) as correctness witness. There are also

[1] Also, we will concentrate only on intraprocedural analysis, though our approach works for interprocedural analysis as well.

```
1  ...
2  <node id="q1">
3  <data key="invariant">( y == x )</data>
4  <data key="invariant.scope">main</data>
5  </node>
6  <edge source="q0" target="q1">
7  <data key="enterLoopHead">true</data>
8  <data key="startline">6</data>
9  <data key="endline">6</data>
10 <data key="startoffset">157</data>
11 <data key="endoffset">165</data>
12 </edge>
13 ...
```

(a) Encoding of an invariant in a GraphML-based correctness witness



(b) Example witness automaton for the program from Fig. 1

Fig. 3. Example of the witness format and automaton; o/w stands for otherwise, i.e., all other possible program transitions

violation witnesses in case a violation has been found, but since we are mainly interested in the invariants, we will focus on correctness witnesses and omit the prefix "correctness" for the rest of the paper.

Such a witness contains a graph representation of an observer automaton. Invariants can be given for nodes if they always hold when the witness automaton is in the corresponding state. The semantics of the witness is given by constructing the product of the witness automaton and the CFA of the program. This might lead to edge cases where the exact semantics depends on how the tool interpreting the witness constructs a CFA from the program, but in practice a witness can be written such that it is mostly robust against those differences. For further details on the semantics of the witness automata we refer the reader the existing literature [12].

There are currently some restrictions on the contents of an invariant: An invariant has to be a valid C expression that can be evaluated to an `int` at the current scope in the program. It may contain conjunctions and disjunctions but no function calls.

2.2 ACSL

Interactive verifiers rely on the user to provide the (non-trivial) invariants for the proof. An example can bee seen in Fig. 2, where the loop invariant has been added as ACSL annotation in line 5. Only when this information is externally

provided (usually by the user), an interactive verifier like Frama-C is able to prove that the assertion in line 11 can never be violated.

Loop annotations are only one of many kinds of annotation in ACSL. For example we can see a function contract in line 1 and an assertion in line 8. These annotations usually represent specifications which the implementation should adhere to, but they can also be seen as invariants, since they should hold for every possible program execution.

The basic building blocks of ACSL annotations are *logic expressions* that represent the concrete properties of the specification, e.g., `a + b > 0` or `x && y == z`. Logic expressions can be subdivided into terms and predicates, which behave similarly as terms and formulas in first-order logic. Basically, logic expressions that evaluate to a boolean value are predicates, while all other logic expressions are terms. The above example `a + b > 0` is therefore a predicate, while `a + b` is a term. We currently support only logic expressions that can also be expressed as C expressions, as they may not be used in a witness otherwise. Finding ways to represent more ACSL features is a topic of ongoing research.

ACSL also features different types of annotations. In this paper we will only present translations for the most common type of annotations, namely function contracts, and the simplest type, namely assertions. Our implementation also supports statement contracts and loop annotations.

All types of ACSL annotations when placed in a C source file must be given in comments starting with an `@` sign, i.e., must be in the form `//@ annotation` or `/*@ annotation */`. ACSL assertions can be placed anywhere in a program where a statement would be allowed, start with the keyword `assert` and contain a predicate that needs to hold at the location where the assertion is placed.

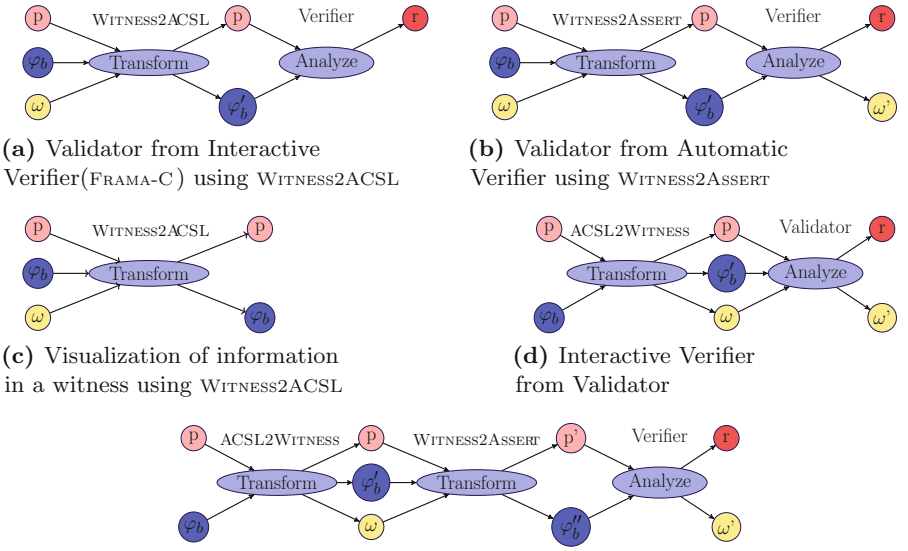
3 A Component Framework for Cooperative Verification

The framework we developed consists of three core components that allow us to improve interaction between the existing tools.

Witness2ACSL acts as transformer that converts a program and a correctness witness given as witness automaton where invariants are annotated to certain nodes, into a program with ACSL annotations.

ACSL2Witness takes a program that contains ACSL annotations, encodes them as invariants into a witness automaton and produces a correctness witness in the standardized GraphML format.

Witness2Assert is mostly identical to Witness2ACSL. The main difference is that instead of adding assertions as ACSL annotations to the program, it actually encodes the semantics of the annotations directly into the program such that automatic verifiers will understand them as additional properties to prove. On the one hand, this component enables us to check the validity of the ACSL annotations for which ACSL2Witness generated a witness, with tools that do not understand the annotation language ACSL. On the other hand, this component is also useful on its own, since it allows us to validate correctness witnesses and give



(a) Validator from Interactive Verifier(Frama-C) using Witness2ACSL

(b) Validator from Automatic Verifier using Witness2Assert

(c) Visualization of information in a witness using Witness2ACSL

(d) Interactive Verifier from Validator

(e) Interactive Verifier from Automatic Verifier

Fig. 4. Graphical visualization of the developed components to improve cooperation; we use the notation introduced in previous work [24]: p represents a program, ϕ_b a behavior specification, ω a witness, and r a verification result

witness producers a better feedback on how their invariants are interpreted and whether they are useful (validator developers can inspect the produced program).

These three components now enable us to achieve cooperation in many different ways. We can utilize a proposed component framework [24] to visualize this as shown in Fig. 4. The use case shown in Fig. 4a is to use Frama-C as a correctness witness validator. This is interesting because it can further reduce the technology bias (the currently available validators are based on automatic verifiers [4,11,13,21], test execution [14], and interpretation [50]). By using Witness2Assert instead of Witness2ACSL as shown in Fig. 4b we can also configure new correctness witness validators that are based on automatic verifiers, similar to what metaval [21] does, only with a different transformer. Figure 4c illustrates the use of Witness2ACSL (or similarly for Witness2Assert) to inspect the information from the witness as annotations in the program code.

The compositional framework makes it possible to leverage existing correctness witness validators and turn them into interactive verifiers that can understand ACSL, as shown in Fig. 4d. Since we also have the possibility now to construct a validator from an automatic verifier (Fig. 4b) we can turn automatic verifiers into interactive ones as depicted in Fig. 4e. While automatic verifiers can already make use of assertions that are manually added to the program, this now also allows us to use other types of high-level annotations like function contracts without having to change the original program.

3.1 Witness2ACSL

To create an ACSL annotated program from the source code and a correctness witness, we first need to extract *location invariants* from the witness, i.e., invariants that always hold at a certain program location (with program locations we refer to the nodes of the CFA here). We can represent location invariants as a tuple (l, ϕ) consisting of a program location l and an invariant ϕ. In general there is no one-to-one mapping between the invariants in the witness and this set of location invariants, since there might be multiple states with different invariants in the witness automaton that are paired with the same program location in the product with the CFA of the program. For extracting the set of location invariants, we calculate this product and then take the disjunctions of all invariants that might hold at each respective location.

3.2 ACSL2Witness

In order to convert the ACSL annotations present in a given program, we transform each annotation into a set of ACSL predicates that capture the semantics of those annotations and use the predicates as invariants in a witness. This mode of operation is based on two observations: Firstly, for a given ACSL annotation it is usually possible to find a number of ACSL assertions that are semantically equivalent to that annotation. For example, a loop invariant can be replaced by asserting that the invariant holds at the loop entry, i.e., before each loop iteration. Secondly, most ACSL assertions are logically equivalent to a valid invariant and can therefore be used in a witness. As mentioned in Sect. 2.2, we currently only support those predicates which can be converted into C expressions, which is a limitation of the witness format and might be lifted in future versions of the format.

3.3 Witness2Assert

This component is very similar to Witness2ACSL. The main difference is that instead of generating ACSL annotations we generate actual C code that encodes the invariants as assertions (i.e., additional reachability properties). This translation is sound since assertions added this way do not hide violations, i.e., every feasible trace that violates the original reachability property in the program before the modification will either still exist or have a corresponding trace that violates the additional reachability properties of the modified program. It is worth mentioning that this is an improvement compared to existing transformations like the one used in MetaVal [21], where the program is resynthesized from the reachability graph and the soundness can therefore easily be broken by a bug in MetaVal's transformation process.

4 Evaluation

We implemented the components mentioned in Sect. 3 in the software-verification framework CPAchecker. In our evaluation, we attempt to answer the following research questions:

- **RQ 1:** Can we construct interactive verifiers from automatic verifiers, and can they be useful in terms of effectiveness?
- **RQ 2:** Can we improve the results of, or partially automate, interactive verifiers by annotating invariants that were computed by automatic verifiers?
- **RQ 3:** Can we construct result validators from interactive verifiers?
- **RQ 4:** Are verifiers ready for cooperation, that is, do they produce invariants that help other verifiers to increase their effectiveness?

4.1 Experimental Setup

Our benchmarks are executed on machines running Ubuntu 20.04. Each of these machines has an Intel E5-1230 processor with 4 cores, 8 processing units, and 33 GB of RAM. For reliable measurements we use BenchExec [20]. For the automatic verifiers, we use the available tools that participated in the ReachSafety category of the 2022 competition on software verification (SV-COMP) in their submission version[2]. Frama-C will be executed via Frama-C-SV [22], a wrapper that enables Frama-C to understand reachability property and special functions used in SV-COMP. Unless otherwise noted we will use the EVA plugin of Frama-C. We limit each execution to 900 s of CPU time, 15 GB of RAM, and 8 processing units, which is identical to the resource limitations used in SV-COMP.

4.2 Benchmark Set with Useful Witnesses

In order to provide meaningful results, we need to assemble an appropriate benchmark set consisting of witnesses that indeed contain useful information, i.e., information that potentially improves the results of another tool.

As a starting point, we consider correctness witnesses from the final runs of SV-COMP 2022 [8,10]. This means that for one verification task we might get multiple correctness witnesses (from different participating verifiers), while for others we might even get none because no verifier was able to come up with a proof. We select the witnesses for tasks in the subcategory ReachSafety-Loops, because this subcategory is focussed on verifying programs with challenging loop invariants. This selection leaves us with 6242 correctness witnesses (without knowing which of those actually contain useful information).

For each of the selected witnesses we converted the contained invariants into both ACSL annotations (for verification with Frama-C) and assertions (for verification with automatic verifiers from SV-COMP 2022). Here we can immediately drop those witnesses that do not result in any annotations being generated, which results in 1931 witnesses belonging to 640 different verification tasks.

[2] https://gitlab.com/sosy-lab/sv-comp/archives-2022/-/tree/svcomp22/2022

Table 1. Impact of cooperation: in each row, a 'consuming' verifier is fed with information from witnesses of our benchmark set; 'Baseline' reports the number of programs that the verifier proved correct without any help; 'Improved via coop.' reports the number of programs that the verifier can prove *in addition*, if the information from the witness is provided

Consuming verifier	Benchmark tasks (434 total)		Projection on programs (230 total)	
	Baseline	Improved via coop.	Baseline	Improved via coop.
2LS	157	179	83	111
UAUTOMIZER	360	47	186	31
CBMC	281	53	142	28
CPACHECKER	300	69	149	53
DARTAGNAN	280	82	139	51
ESBMC	239	133	121	76
GAZER-THETA	266	118	135	64
GOBLINT	38	106	21	47
UKOJAK	191	134	97	76
KORN	183	46	98	27
PESCO	180	162	87	99
PINAKA	258	105	127	59
SYMBIOTIC	349	51	174	32
UTAIPAN	334	65	172	37
VERIABS	343	31	186	28
FRAMA-C	211	31	105	20

We then run each verifier for each program where annotations have been generated, once with the original, unmodified program, and n times with the transformed program for each of the n witnesses. This allows us determine whether any improvement was achieved, by looking at the differences between verification of the unmodified program versus verification of a program that has been enhanced by information generated from some potentially different tool. Using this process, we further reduce our benchmark set of witnesses to those that are useful for at least one of the verifiers and thus enable cooperation. This leads to the final set of 434 witnesses that evidently contain information that enables cooperation between verifiers. These witnesses correspond to 230 different programs from the SV-Benchmarks repository (https://github.com/sosy-lab/sv-benchmarks). We made this benchmark set available to the community in a supplementary artifact of this paper [23].

4.3 Experimental Results

RQ 1. For the first research question, we need to show that we can construct interactive verifiers from automatic verifiers, and that they can be useful in terms of effectiveness. By "interactive verifier", we mean a verifier that can verify

more programs correct if we feed it with invariants, for example, by annotating the input program with ACSL annotations. Using our building blocks from Sect. 3, an interactive verifier can be composed as illustrated in Fig. 4e (that is, configurations of the form ACSL2Witness|Witness2Assert|Verifier). For a meaningful evaluation we need a large number of annotated programs, which we would be able to get if we converted the witnesses from SV-COMP using Witness2ACSL in advance. But since the first component ACSL2Witness in Fig. 4e essentially does the inverse operation, we can generalize and directly consider witnesses as input, as illustrated in Fig. 4b (that is, configurations of the form Witness2Assert|Verifier).

Now we look at the results in Table 1: The first row reports that cooperation improves the verifier 2ls in 179 cases, that is, there are 179 witnesses that contain information that helps 2ls to prove a program that it could not prove without the information. In other words, for 179 witnesses, we ran Witness2Assert to transform the original program to one in which the invariants from the witness were written as assertions, and 2ls was then able to verify the program. Since there are often several witnesses for the same program, 2ls verified in total 111 unique unique programs that it was not able to verify without the annotated invariants as assertion.

In sum, the table reports that many programs that could not be proved by verifiers when ran on the unmodified program, could be proved when the verifier was given the program with invariants. Since we were able to show the effect using generated witnesses, it is clear that manually provided invariants will also help the automatic verifiers to prove the program. We will continue this argument in Sect. 4.4.

RQ 2. For the second research question, we need to show that our new design can improve the results of interactive verifiers by annotating invariants that were computed by automatic verifiers. Using our building blocks from Sect. 3, we assemble a construction as illustrated in Fig. 4a (i.e., configurations of the form Witness2ACSL|Verifier). We take a program and a witness and transform the program to a new program that contains the invariants from the witness as ACSL annotations.

Let us consider the last row in Table 1: Frama-C is able to prove 20 programs correct using invariants from 31 witnesses. Those 31 witnesses were computed by automatic verifiers, and thus, we can conclude that our new design enables using results of automatic verifiers to help the verification process of an interactive verifier.

RQ 3. For the third research question, we need to show that we can construct result validators from interactive verifiers and that they can effectively complement existing validators. A results validator is a tool that takes as input a verification task, a verdict, and a witness, and confirms or rejects the result. In essence, due to the modular components, the answer to this research question can be given by the same setup as for RQ 2: If the interactive verifier (Frama-C) was able to prove the program correct, then it also has proved that the invariants provided by the witnesses were correct, and thus, the witness should be confirmed. Frama-C has confirmed 31 correctness witnesses.

Table 2. Proof of cooperation: for each 'producing' verifier, we report the number of correctness witnesses that help another verifier to prove a program which it otherwise could not; we also list the number of cases where this cooperation was observed (some witnesses improve the results of multiple verifiers); we omit producers without improved results

Producing verifier	Useful witnesses	Cases of cooperation
2LS	1	1
CBMC	20	22
CPACHECKER	148	533
GOBLINT	2	3
GRAVES-CPA	151	823
KORN	10	15
PESCO	78	271
SYMBIOTIC	5	10
UAUTOMIZER	19	70
Sum	434	1748

New validators that are based on a different technology are a welcome complement because this reduces the technology bias and increases trust. Also, the proof goals for annotated programs might be interesting for verification engineers to look at, even or especially when the validation does not succeed completely.

RQ 4. For the fourth research question, we report on the status of cooperation-readiness of verifiers. In other words, the question is if the verifiers produce invariants that help other verifiers to increase their effectiveness.

In Table 2 we list how many useful witnesses each verifier contributed to our benchmark set of useful witnesses. The results show that there are several verifiers that produce significant amounts of witnesses that contain invariants that help to improve results of other verifiers.

4.4 Case Study on Interactive Verification with Manual Annotations

So far, we tested our approach using information from only the SV-COMP witnesses. For constructing interactive verifiers, we would also like to evaluate whether our approach is useful if the information is provided by an actual human in the form of ACSL annotations.

ACSL Benchmark Set. To achieve this, we need a benchmark set with tasks that contain sufficient ACSL annotations and also adhere to the conventions of SV-COMP. Since to our knowledge such a benchmark set does not exist yet, we decided to manually annotate assertions and loop invariants to the tasks from the SV-Benchmarks collection ourselves. While annotating all of the benchmark tasks is out of scope, we managed to add ACSL annotations to 125 tasks from the ReachSafety-Loops subcategory. This subcategory is particularly relevant, since it contains a selection of programs with interesting loop invariants. The loop invariants we added are sufficient to proof the tasks correct in a pen-and-paper,

Table 3. Case study with 125 correct verification tasks where sufficient, inductive loop invariants are manually annotated to the program; we either input these to Frama-C or automatically transform the annotations into witnesses and try to validate these witnesses using CPAchecker's k-induction validator (with k fixed to 1); the listed numbers correspond to the number of successful proofs in each of the sub-folders; we also list the number of successful proofs if no invariants are provided to the tools

Subfolder	Tasks	Frama-C		k-induction	
		with invs.	without invs.	with invs.	without invs.
loop-acceleration	17	3	1	11	4
loop-crafted	2	0	0	2	2
loop-industry-pattern	1	0	0	1	1
loop-invariants	8	3	0	8	0
loop-invgen	5	0	0	2	0
loop-lit	11	6	0	10	2
loop-new	5	1	0	5	2
loop-simple	6	6	0	1	1
loop-zilu	20	9	0	19	7
loops	23	13	6	17	15
loops-crafted-1	27	0	0	12	1
total	125	41	7	88	35

Hoare-style proof. Our benchmark set with manually added ACSL annotations is available in the artifact for this paper [23].[3]

Construction of an Interactive Verifier. With our ACSL benchmark set, we can now convert a witness validator into an interactive verifier as depicted in Fig. 4d. For the validator we use CPAchecker, which can validate witnesses by using the invariants for a proof by k-induction. By fixing the unrolling bound of the k-induction to $k = 1$, this will essentially attempt to prove the program correct via 1-induction over the provided loop invariants. If we do not fix the unrolling bound, the k-induction validation would also essentially perform bounded model checking, so we would not know whether a proof succeeded because of the provided loop invariants or simply because the verification task is bounded to a low number of loop iterations.

Since this 1-induction proof is very similar to what Frama-C's weakest-precondition analysis does, we can directly compare both approaches. As some tasks from the benchmark set do not require additional invariants (i.e., the

[3] Our benchmark set is continuously updated and can also be found at: https://gitlab.com/sosy-lab/research/data/acsl-benchmarks

property to be checked is already inductive) we also analyze how both tools perform on the benchmark set if we do not provide any loop invariants.

The experimental setup is the same described in Sect. 4.1, except that we use a newer version of FRAMA-C-SV in order to use the weakest-precondition analysis of FRAMA-C. The results are shown in Table 3, which lists the number of successful proofs by subfolder. We can observe that both FRAMA-C and our constructed interactive verifier based on CPACHECKER can make use of the information from the annotations and prove significantly more tasks compared to without the annotated loop invariants. This shows that the component described in Fig. 4d is indeed working and useful.

5 Conclusion

The verification community integrates new achievements into two kinds of tools: interactive verifiers and automatic verifiers. Unfortunately, the possibility of cooperation between the two kinds of tools was left largely unused, although there seems to be a large potential. Our work addresses this open problem, identifying witnesses as interface objects and constructing some new building blocks (transformations) that can be used to connect interactive and automatic verifiers. The new building blocks, together with a cooperation framework from previous work, make it possible to construct new verifiers, in particular, automatic verifiers that can be used interactively, and interactive verifiers that can be fed with information from automatic verifiers: Our new program transformations translate the original program into a new program that contains invariants in a way that is understandable by the targeted backend verifier (interactive *or* automatic). Our combinations do not require changes to the existing verifiers: they are used as 'off-the-shelf' components, provided in binary form.

We performed an experimental study on witnesses that were produced in the most recent competition on software verification and on programs with manually annotated loop invariants. The results show that our approach works in practice: We can construct various kinds of verification tools based on our new building blocks. Instrumenting information from annotations and correctness witnesses into the original program can improve the effectivity of verifiers, that is, with the provided information they can verify programs that they could not verify without the information. Our results have many practical implications: (a) automatic verification tools can now be used in an interactive way, that is, users or other verifiers can conveniently give invariants as input in order to prove programs correct, (b) new validators based on interactive verifiers can be constructed in order to complement the set of currently available validators, and (c) both kinds of verifiers can be connected in a cooperative framework, in order to obtain more powerful verification tools. This work opens up a whole array of new opportunities that need to be explored, and there are many directions of future work. We hope that other researchers and practitioners find our approach helpful to combine existing verification tools without changing their source code.

Data-Availability Statement. The witnesses that we used are available at Zenodo [10]. The programs are available at Zenodo [9] and on GitLab at https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp22. We implemented our transformations in the verification framework CPAchecker, which is freely available via the project web site at https://cpachecker.sosy-lab.org. A reproduction package for our experimental results is available at Zenodo [23].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

Acknowledgment. We thank Nikolai Kosmatov for an inspiring and motivating discussion at the conference ISoLA 2018 on the necessity to combine automatic and interactive verification.

References

1. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141 (2019). https://doi.org/10.1109/ASE.2019.00121
2. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The key tool. Software and Systems Modeling **4**(1), 32–54 (2005). https://doi.org/10.1007/s10270-004-0058-x
3. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making software verification tools really work. In: Proc. ATVA. pp. 28–42. LNCS 6996, Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_3
4. Ayaziová, P., Chalupa, M., Strejček, J.: Symbiotic-Witch: A Klee-based violation witness checker (competition contribution). In: Proc. TACAS (2). pp. 468–473. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_33
5. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with Slam. Commun. ACM **54**(7), 68–76 (2011). https://doi.org/10.1145/1965724.1965743
6. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
7. 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
8. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
9. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
10. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5838498
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351

12. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. (2022). https://doi.org/10.1145/3477579
13. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
14. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
15. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
16. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
17. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
18. 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
19. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11
20. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
21. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
22. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 429–434. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_26
23. Beyer, D., Spiessl, M., Umbricht, S.: Reproduction package for SEFM 2022 article 'Cooperation between automatic and interactive software verifiers'. Zenodo (2022). https://doi.org/10.5281/zenodo.6541544
24. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
25. 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
26. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
27. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM. pp. 132–146. LNCS 7436, Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13

28. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
29. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2). pp. 38–47. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
30. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
31. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE. pp. 100–114. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7
32. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. arXiv:2010.05812v2, arXiv (January 2020). https://doi.org/10.48550/arXiv.2010.05812
33. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: Overview and VerifyThis competition. Int. J. Softw. Tools Technol. Transf. **17**(6), 677–694 (2015). https://doi.org/10.1007/s10009-014-0308-3
34. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: VerifyThis: Verification competition with a human factor. In: Proc. TACAS. pp. 176–195. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_12
35. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: Proc. FMICS. pp. 3–69. LNCS 12327, Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
36. Gerrard, M.J., Dwyer, M.B.: Comprehensive failure characterization. In: Proc. ASE. pp. 365–376. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115649
37. Gerrard, M.J., Dwyer, M.B.: ALPACA: A large portfolio-based alternating conditional analysis. In: Atlee, J.M., Bultan, T., Whittle, J. (eds.) Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019. pp. 35–38. IEEE / ACM (2019). https://doi.org/10.1109/ICSE-Companion.2019.00032
38. Hales, T.C., Harrison, J., McLaughlin, S., Nipkow, T., Obua, S., Zumkeller, R.: A revision of the proof of the Kepler conjecture. Discret. Comput. Geom. **44**(1), 1–34 (2010). https://doi.org/10.1007/s00454-009-9148-4
39. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
40. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **1610**(06229) (October 2016)
41. Heule, M.J.H.: Schur number five. In: Proc. AAAI. pp. 6598–6606. AAAI Press (2018)
42. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.S.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. Int. J. Softw. Tools Technol. Transfer **16**(5), 457–464 (2014). https://doi.org/10.1007/s10009-014-0337-y
43. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. NFM. pp. 41–55. LNCS 6617, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
44. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: Proc. SPIN. pp. 30–39. ACM (2014). https://doi.org/10.1145/2632362.2632372

45. Jakobs, M.C., Wehrheim, H.: Programs from proofs: A framework for the safe execution of untrusted software. ACM Trans. Program. Lang. Syst. **39**(2), 7:1–7:56 (2017). https://doi.org/10.1145/3014427
46. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proc. LPAR. pp. 348–370. LNCS 6355, Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
47. Necula, G.C.: Proof-carrying code. In: Proc. POPL. pp. 106–119. ACM (1997). https://doi.org/10.1145/263699.263712
48. Richter, C., Hüllermeier, E., Jakobs, M.-C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
49. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
50. Švejda, J., Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NitWit. In: Proc. TACAS. pp. 40–57. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_3

A Unifying Approach for Control-Flow-Based Loop Abstraction

Dirk Beyer, Marian Lingsch Rosenfeld, and Martin Spiessl

LMU Munich, Munich, Germany

Abstract. Loop abstraction is a central technique for program analysis, because loops can cause large state-space representations if they are unfolded. In many cases, simple tricks can accelerate the program analysis significantly. There are several successful techniques for loop abstraction, but they are hard-wired into different tools and therefore difficult to compare and experiment with. We present a framework that allows us to implement different loop abstractions in one common environment, where each technique can be freely switched on and off on-the-fly during the analysis. We treat loops as part of the abstract model of the program, and use counterexample-guided abstraction refinement to increase the precision of the analysis by dynamically activating particular techniques for loop abstraction. The framework is independent from the underlying abstract domain of the program analysis, and can therefore be used for several different program analyses. Furthermore, our framework offers a sound transformation of the input program to a modified, more abstract output program, which is unsafe if the input program is unsafe. This allows loop abstraction to be used by other verifiers and our improvements are not 'locked in' to our verifier. We implemented several existing approaches and evaluate their effects on the program analysis.

Keywords: Software verification · Program analysis · Loop abstraction · Precision adjustment · Counterexample-guided abstraction refinement · CPAchecker

1 Introduction

Software programs are among the most complex systems that mankind produces. Programs tend to have a complex state space and hence verifying the correctness of software programs is a difficult task. Abstraction is a key ingredient to every successful approach to prove the correctness of large programs. Let us look at a few examples: Constant propagation [21] abstracts from concrete values for a variable if the value of the variable is not constant. Counterexample-guided abstraction refinement (CEGAR) [14] is an algorithm to incrementally refine the level of abstraction until the abstract model is detailed enough to prove the correctness, while the abstract model is still coarse enough to make the analysis feasible. Predicate abstraction [18,20] uses an abstract domain where the abstract state is described as a combination of predicates from a certain

B.-H. Schlingloff and M. Chai (Eds.): SEFM 2022, LNCS 13550, pp. 3–19, 2022.
https://doi.org/10.1007/978-3-031-17108-6_1

given precision [8] (a set of predicates). The precision is refined with CEGAR by adding new predicates to the precision. Shape analysis [25] abstracts from concrete data structures on the heap and stores only their shape for the analysis.

Finally, *loop abstraction* is a technique to abstract the behavior of a program with a loop in such a way that the correctness of the abstract program implies the correctness of the original program. There are several approaches for loop abstraction proposed in the literature [15,16,19,22]. While we will concentrate on reachability here, this technique can also be applied to other properties.

We contribute a formalism that treats loop abstraction as an abstraction in the sense of CEGAR: The precision is a choice of a certain approach to loop abstraction (level of abstraction of the loop). If the abstract model of the program defined by this precision (= loop abstraction) is too coarse to prove correctness, then we refine the abstract model by setting the precision to a different (more precise) loop abstraction.

Example. Let us consider the small program in Fig. 1a. The program uses one variable `x`, which is initialized with some large, even value and decreased by 2 in a loop. The specification requires that the value of `x` is even after the loop terminates. It is easy for a human to see that an even number, decreased by an even number, always yields an even number, no matter how often this is done. In other words, we discover the invariant that `x` is even and check if it is preserved. However, in this example there exists an even simpler invariant: The data type of `x` is unsigned int, which means values greater or equal to zero. The control flow cannot leave the loop as long as `x` is greater than 0. Once the control flow leaves the loop, we know that the value is 0, and thus, even. The loop-exit condition, together with the above argument, implies the specification. A program analysis that cannot discover this (e.g., bounded model checking, explicit-value analysis, interval analysis) has to unroll the loop many times.

But we can construct the loop abstraction in Fig. 1b, which executes the new body only if the loop condition `x > 0` is fulfilled, and the new body models all behaviors that occur when the original program enters the loop. The new body havocs (sets to an arbitrary value) the variable `x`. Then it constrains the values of `x` by blocking the further control flow if the loop condition still holds, i.e., the original program would stay in the loop. Surprisingly, since the loop-exit condition now implies the specification, this overapproximation of the original program still satisfies the specification.

Contributions. This paper makes the following contributions:

- We propose a framework that can express several existing approaches for loop abstraction and makes it possible to compare those different approaches.
- The framework allows to switch dynamically, on-the-fly, between different loop-abstraction techniques, selecting different abstraction levels.
- The framework is independent from the underlying abstract domain of the program analysis. The loop abstractions work using transformations of the control flow. Once implemented, a strategy for loop abstraction is applicable to several abstract domains.

A Unifying Approach for Control-Flow-Based Loop Abstraction 5

```
unsigned int x = 0x0ffffff0;
while (x > 0) {
  x -= 2;
}
assert(!(x % 2)));
```

(a) Original program

```
unsigned int x = 0x0ffffff0;
if (x > 0) {
  x = nondet_uint();
  if (x > 0) {
    return 0;
  }
}
assert(!(x % 2)));
```

(b) Havoc abstraction

```
unsigned int x = 0x0ffffff0;
if (x > 0) {
  long long iterations = x/2;
  x -= 2*iterations;
  if (x > 0) {
    x -= 2;
  }
}
assert(!(x % 2)));
```

(c) Constant extrapolation

```
unsigned int x = 0x0ffffff0;
if (x > 0) {
  x = nondet_uint();
  if (x <= 0) {
    return 0;
  }
  x -=2;
  if (x > 0) {
    return 0;
  }
}
assert(!(x % 2)));
```

(d) Naive abstraction

Fig. 1. Application of various loop abstraction strategies on the benchmark program `simple_4-2.c` from the SV-Benchmarks set; only the body of the main function is shown here

- We export the modified C program, such that the loop-abstraction techniques can be used by other verifiers.
- The framework is publicly available as an extension of the open-source verification framework CPAchecker.
- We evaluate the effectiveness and efficiency of the framework on a benchmark set from the publicly available collection of verification tasks SV-Benchmarks, and compare it with state-of-the-art tools.

Related Work. In the following we discuss the most related existing approaches.

Loop Acceleration. As this is an obvious way to speed up verification, many different approaches have been proposed to calculate the effects of a loop execution [17,19,26]. We present only a very basic form where we accelerate variables that are incremented by a fixed value in loops with a known number of iterations, since our interest is rather into gaining insights into how different existing approaches can be combined to further improve their usefulness. As such we are interested in implementing other approaches for loop acceleration as strategies into our framework, rather than coming up with new ways of accelerating single loops.

Loop Abstraction. While loop acceleration is useful also in other areas, e.g., for compiler optimizations, verifiers have the possibility of using loop abstractions (i.e., overapproximatons) instead, for aiding the generation of correctness proofs. Since loop abstraction is closely related to invariant generation, and this is the main challenge in software verification, there is a large body of literature. We will therefore look at only those publications that also make use of the idea to encode the abstractions into the source code. The abstraction techniques we describe in this paper are taken taken from existing publications [15,16]. As with loop accelerations, our goal is not to invent new strategies, but rather investigate how existing strategies can be combined. Also VeriAbs [1] uses a variety of loop-abstraction techniques, but only statically generates a program that is then checked by a third-party verifier. As fallback, the original program is verified.

Encoding Loop Abstractions into the Program. We found one publication that also encodes loop accelerations into a modified program [23]. Here, the accelerated loop variant is added in such a way that the alternative code will be entered based on non-deterministic choice. The main motivation is to investigate how this can create synergies with invariant generation, i.e., whether existing invariant generators can be improved by also providing the results of the acceleration in the program. Compared to that, our approach is more general, as we also consider overapproximating loop abstractions. Instead of non-deterministic choice, we present an approach to determine which strategies to use automatically using CEGAR.

2 Preliminaries

We quickly introduce some notation and common concepts that will later be used in Sect. 3.1.

Program Semantics. For simplicity we will consider a programming language where the set *Ops* of possible program operations consists of simple assignments and assumptions. We represent the programs as control-flow automata (CFA). A CFA $C = \{L, l_0, G\}$ consists of a set L of program locations (modeling the progam counter), an initial program location l_0, and a relation $G \subseteq L \times Ops \times L$ that describes the control-flow edges (each modeling the flow from one program location via a program operation to a successor program location). The concrete semantics of such a CFA is given by the (labeled) transition relation $\rightarrow \subseteq C \times G \times C$ over the set C of concrete program states. We will write $c_1 \xrightarrow{g} c_2$ if the concrete state c_2 can be reached from c_1 via the control-flow edge $g \in G$.

Program Analysis. Our approach will work for many different kinds of program analysis. Typically, a program analysis is characterized by some abstract domain D that defines a set E of abstract states as well as an abstract transfer relation $\rightsquigarrow \subseteq E \times G \times E$, which determines which abstract states can be reached from the initial state $e_0 \in E$. One common way to design a program analysis is to determine the set of reachable abstract states by keeping track of a set `reached` $\subseteq E$ of

already reached abstract states and a set (or list) `waitlist` $\subseteq E$ of abstract states that still need to be explored.[1]

CEGAR. Whenever a program analysis finds a counterexample, there are two possibilities. Either this turns out to correspond to an actual execution trace of the original program, and we have shown that the program violates the specification, or the counterexample is infeasible, meaning that it is only found because the abstraction level of the analysis is too coarse. This has led to the development of *counterexample-guided abstraction refinement*, or CEGAR for short [14]. The idea here is that one can extract information from the counterexample with which the abstract domain can be refined. For example with predicate abstraction[2], one can use the counterexample to compute predicates that —if tracked— rule out the infeasible counterexample. In order to formalize CEGAR, we will introduce the refinement operator:

$$\texttt{refine} : (\texttt{reached}, \texttt{waitlist}) \mapsto (\texttt{reached'}, \texttt{waitlist'})$$

Once an infeasible counterexample is found, the refinement operator is called with the current set of reached abstract states and the waitlist. This operator then extracts information from its inputs and returns a new set of reached states and a new waitlist which will then be used for further state-space exploration. In case the counterexample is feasible, the refinement operator will not remove the violation state(s) from the set of reached abstract states, which signals that the analysis found a bug and can terminate.

3 Loop Abstractions

We propose the approach of multi-strategy program analysis, which enables one tool to use several different loop-abstraction strategies simultaneously in one state-space exploration. In the following, we will first look at the theory behind loop abstractions and some practical examples for such strategies. After that, we will introduce our CEGAR refinement approach for loop abstractions in Sect. 3.2.

3.1 Theory

For verification, we usually use overapproximations if the goal is to find a proof of correctness. For loop control flow, such an overapproximation is called a *loop abstraction*, while precise methods are called *loop acceleration*. Whenever it is not important whether the technique is precise or overapproximating, we will just refer to the techniques as loop abstraction.

It is common to apply loop abstractions by replacing the loop statement S with some alternative program statement S' [1,23]. Intuitively, it is often clear whether this will overapproximate the program behavior, but we can also formalize this

[1] In the literature, this is also know as a worklist algorithm [24]; here we will adhere to the terminology used in the Handbook of Model Checking [6].

using strongest postconditions. We write $sp(S, P)$ for the strongest postcondition of a program statement S and a predicate P. Assume we have a program statement S that contains a loop, i.e., $S =$ `while (C) do B`, where the body `B` inside S may itself contain loops. For a loop abstraction, the goal is to find an alternative program statement S' such that $\{P\}S\{sp(S', P)\}$ is a valid Hoare triple. If this requirement is fulfilled, then we can soundly replace S by S' in the program for the purpose of verification. In other words, S' is an abstraction of S if $sp(S, P) \Rightarrow sp(S', P)$. It is possible to find such rewriting schemes for a loop without knowing the exact form of the loop. This is best shown by two examples.

Havoc Abstraction. Let us look at the rather simple loop abstraction that served as example in Sect. 1, which we call *havoc abstraction*. Here we replace the loop `while C do B` by a havoc statement `havoc(mod(B))` that is guarded in such a way to ensure it is only executed if the loop condition holds, and after it is executed, the loop condition does not hold anymore. The havoc statement discards any information about the values of a set of variables. Here we use the set `mod(B)` of variables that are modified in the loop body `B`. We denote the strongest postcondition of this havoc statement by $H_{B,P} = sp(\texttt{havoc(mod(B))}, P)$. We can easily prove soundness of the havoc abstraction by establishing that $H_{B,P}$ is actually a loop invariant and therefore the Hoare triple $\{P\}$ `while C do B` $\{H_{B,P} \wedge \neg C\}$ holds.[2]

It is obvious that we can find an alternative statement S' for the while-loop that has the same post condition:

$$sp(\texttt{havoc(mod(B));assume(!C)}, P) = H_{B,P} \wedge \neg C$$

We therefore have found a statement whose strongest post is an overapproximation of the strongest post of the while loop.

Naive Abstraction. Another way to abstract a loop is the so-called naive loop abstraction [16]. An application to the example program from Fig. 1a is shown in Fig. 1d. Here one assigns non-deterministic values to all the variables that are modified in the loop (provided the loop condition holds when reaching the loop). Then the loop body is executed once, after which the negated loop condition is added as assumption. This essentially encodes the information that if the loop was entered, there is a "last" iteration of the loop after which the loop condition does not hold anymore and the loop therefore terminates. This is overapproximating the behavior of the original loop, since a loop, in general, is not guaranteed to terminate. From the Hoare proof of the naive abstraction, we get that $sp(B, C \wedge H_{B,P}) \vee P$ is an invariant of the while loop.[3]

The postcondition $(sp(B, C \wedge H_{B,P}) \vee P) \wedge \neg C$ that is shown in the proof is also the post condition of the alternative code for the loop described above:

$$sp(\texttt{if C then \{havoc(mod(B));assume(C);B;assume(!C)\}}, P) = \\ (sp(B, C \wedge H_{B,P}) \vee P) \wedge \neg C$$

[2] Proof can be found at: https://www.sosy-lab.org/research/loop-abstraction/
[3] Proof can be found at: https://www.sosy-lab.org/research/loop-abstraction/

Observations. We can make three interesting observations by looking at these proofs. Firstly, we eliminated the outermost loop from the statement S, at the cost of overapproximation. If this can be achieved iteratively until no loops are left, the resulting overapproximation can be quickly checked by a (possibly bounded) model checker, as no loops need to be unrolled anymore.

Secondly, in the proof we actually used an invariant for applying the while-rule. Every loop-abstraction strategy can therefore be seen as a way to generate invariants for a loop based on some structural properties of the loop. In the example of the havoc abstraction, we used the fact that for a precondition P, $H_{B,P}$ is always preserved by a loop (provided there is no aliasing). The invariant depends on the precondition P, so for every precondition with which the loop can be reached, the loop abstraction yields a different state invariant. Without knowing P it can only be expressed as a transition invariant that may refer to the "old" values of variables before entering the loop. One can compute a state invariant by assuming the most general precondition $P = \mathit{true}$, but this will often eliminate most of the useful information from the invariant. As transition invariants can often be expressed precisely by program statements, this explains why for loop abstraction, we choose to replace the loop statement with alternative program statements that capture the corresponding transition invariant. This invariant view on loop abstraction works in both ways, meaning that if an invariant is provided for a loop, we can use this invariant for abstracting the loop. It is even possible to construct an inductive proof this way, i.e., transforming the loop in such a way that model checking of the resulting program will essentially carry out a combined-case (k-)inductive proof [15].

The third observation is that the invariant of one loop abstraction might sometimes imply the invariant of another loop abstraction. This is the case in the two examples: the invariant for havoc loop abstraction is implied by the invariant we use in the naive loop abstraction. This means we can build a hierarchy, where naive loop abstraction overapproximates the original loop, and havoc abstraction overapproximates naive abstraction. We will exploit the idea of this abstraction hierarchy later in Sect. 3.2 for an abstraction-refinement scheme.

Constant Extrapolation. For loops where we can calculate the exact number of iterations as well as the final values of the variables assigned in the loop (e.g., because the loop is linear or otherwise easily summarizable) we can simply accelerate the loop by replacing it with assignment statements for the final variable values. The application of constant extrapolation to the program from Fig. 1a is shown in Fig. 1c. For the program in Fig. 2, this would replace the loop with a statement that increments the variable `i` by `N`. For programs like the one shown in Fig. 3 that contains a potential property violation inside the loop, one has to be careful to preserve those violations that can occur in any of the loop iterations.

3.2 Combining Strategies for Loop Abstraction

In Sect. 3.1 we already introduced various ways to abstract loops, which we will in the following refer to as strategies. Intuitively, a strategy is a way to compute an abstraction of a loop that is helpful to verify a program.

Since there are often many different strategies that could be applied to a loop in the program, we need to make some choice about which strategies to use. The simplest approach that is used in the state-of-the-art verification tool VeriAbs is to choose the most promising that can be applied for each loop, generate a program where the loops are rewritten according to these strategies, and hand this program over to a (possibly bounded) verifier for verification.

This has the downside that in cases where the program contains multiple loops, the chosen approximations might be either not abstract enough for the verifier to calculate the proof efficiently or too abstract, such that the proof of the property does not succeed. Choosing a good abstraction level is one of the key challenges in software verification. One successful way how this can be solved is counterexample-guided abstraction refinement (CEGAR) [14].

Our idea is therefore to use CEGAR in order to refine the abstraction of the program dynamically during the program analysis, which allows us to try multiple strategies for the same loop in the program, and even different strategies for the same loop at different locations in the state-space exploration. Because a program analysis operates on the CFA, and loop abstractions correspond to transition invariants that can often be expressed naturally as a sequence of progam instructions, we choose to encode the loop abstractions directly into the CFA. This allows us to realize the CEGAR approach for loop abstractions independently of the details of the exact program analysis that is used.

Encoding of Strategies. We encode strategies that are to be considered directly into the CFA of the program. The CFA for a program statement S such as a loop has a unique entry node α and a unique exit node ω. The application of a strategy to this statement results in the statement S' and a CFA with an entry node α' and an exit node ω'. We attach the CFA for the statement S' of a strategy with two dummy transitions $\alpha \to \alpha'$ and $\omega' \to \omega$, as depicted in Fig. 4. Here, we explicitly denoted the entry edge for the strategy application with the keyword `enter` followed by an identifier that makes clear which strategy was applied (here, `h` stands for havoc). The resemblance to function call and return edges is not a coincidence. By keeping track of the currently entered strategy applications, e.g. in form of a stack, it will always be clear which parts of the execution trace correspond to executions in the original program, and which parts are part of some —potentially overapproximating— strategy application. For nested loops, we can apply the strategies starting from the inner-most loop and construct alternatives in the CFA for all possible strategy combinations.

A CFA that is augmented with strategies in this way contains all program traces of the original program, and can non-deterministically branch into any of the strategy applications. In order to make use of this modified CFA, the analysis needs to be able to distinguish between the original control flow and nodes in the CFA at which we start to apply a particular strategy. The important nodes for this are the entry nodes for each of the strategy applications, so we augment the modified CFA $C = (L, l_{init}, G)$ with a strategy map $\sigma : L \to N$ that maps each CFA node $l \in L$ to a strategy identifier $\sigma(l) \in N$ and call the resulting tuple $\Gamma = (C, \sigma)$ a *strategy-augmented CFA*. The set N of strategy identifiers

```
1 void main() {
2   int i = 0;
3   while (i<N) {
4     i=i+1;
5   }
6   assert (i==N);
7 }
```

Fig. 2. Example program 1: potential property violation outside the loop

```
1 void main() {
2   int i = 0;
3   while (i<N) {
4     i=i+2;
5     assert(i%2==0);
6   }
7 }
```

Fig. 3. Example program 2: potential property violation inside the loop

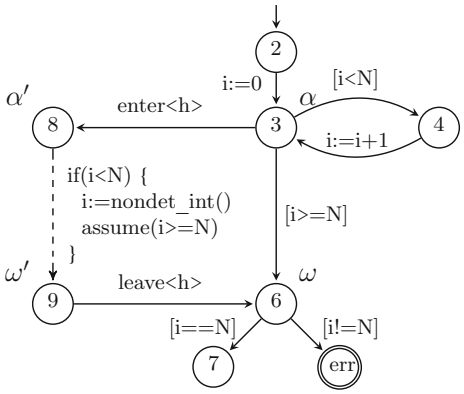


Fig. 4. CFA C of example program from Fig. 2, with an additional application of the havoc strategy

contains a special strategy b, which we call the base strategy. The strategy map σ maps the entry node for each strategy application to the corresponding strategy's identifier, while all other nodes are mapped to the base strategy b.

In a program analysis, we can now use the strategy map for selecting exactly the transitions we want to follow. For example, we can always follow the original program by excluding all transitions to CFA nodes with an associated strategy identifier that is different from the base strategy. By using a more general selection function, we have fine-grained control over which strategies we are applying, which we will describe in the following. As this modifies only the transition relation of the state-space exploration, it can be seamlessly applied to a wide variety of such algorithms.

Selection of Strategies. At any node l in an augmented CFA, we can calculate the set $A \subseteq N$ of available strategies as:

$$A = \{\sigma(l') \mid \exists g \in G : l \xrightarrow{g} l'\}$$

In order to define which strategies should be applied (e.g., because others overapproximate too much and lead to false alarms), we define a precision set $\pi_S \subseteq N$

which we call the *strategy precision*. This precision can be tracked along each abstract state of the program analysis. In practice this precision is tracked for each program location separately, but for simplicity of presentation, we will only consider a global precision here. Semantically the precision expresses which strategies are allowed to be taken from the current abstract state. We can now express different selection approaches by defining a function $\texttt{select} : \mathcal{P}(N) \times \mathcal{P}(N) \to \mathcal{P}(N)$, which needs to fulfill the property $\texttt{select}(A, \pi_S) \subseteq A \cap \pi_S$.

The exact choice of the function `select` depends on the use case and the set of available strategies. One possibility which we will use is to define a partial order $\sqsubseteq$ over the set of available strategy identifiers, and derive the selection function in the following way:

$$\texttt{select}(A, \pi_S) = \{s \in A \cap \pi_s \mid \nexists s' \in A \cap \pi_s : s \sqsubseteq s'\} \tag{1}$$

Such a partial order can be based on the invariant hierarchy of the loop-abstraction strategies, as motivated in Sect. 3.1. It is of course not guaranteed that deciding whether one invariant implies the other is actually decidable. But depending on the strategies considered, one can also just take some design decisions regarding the partial order. In general it is desirable to have the base strategy as greatest lower bound, since as long as only overapproximation is considered, this is the most precise strategy.

The selection function above will return the most abstract strategies, i.e., that overapproximate most. Once we rule those out by removing their strategy identifier from the precision, more and more precise strategies will be returned.

CEGAR Refinement Chaining. We can now define the refinement operator `refine` for precision-based loop acceleration on top of any refiner of an existing analysis, which we will call the wrapped refiner $\texttt{refine}_{\mathbb{W}}$. This can be done by composing the refinement operator $\texttt{refine}_{\mathbb{W}}$ with the strategy-refinement operator $\texttt{refine}_{\mathbb{S}}$, which updates the strategy precision with information from the error path:

$$\texttt{refine} = \texttt{refine}_{\mathbb{S}} \circ \texttt{refine}_{\mathbb{W}} \tag{2}$$

Since the wrapped refinement operator is executed first, it gets the possibility to remove all error states from the reached set, in which case $\texttt{refine}_{\mathbb{S}}$ has nothing to do and will just return its inputs. If there are still error states left in the reached set after $\texttt{refine}_{\mathbb{W}}$ was executed, this means that the inner refinement has discovered a feasible error path for the augmented CFA. Now it depends on whether any overapproximating strategies were used on the error paths that are present in the reached set. If there are none, then the error path is indeed also present in the real program and $\texttt{refine}_{\mathbb{S}}$ returns the reached set with the error state(s), indicating that a bug has been found. An example for this would be the case where only constant extrapolation has been used along the path. If there are overapproximating strategies such as the havoc abstraction on an error path, we can adapt the strategy precision in order to rule out that we will find the same error path again after the refinement. For that, we locate the first abstract state on the path whose successor enters an overapproximating strategy (the so-called

pivot state) and adapt the strategy precision such that this strategy can not be selected in the future. We then remove all (transitive) successor abstract states of that pivot state from the set of reached abstract states.

Example. The chaining of the refinement operators is best visualized by looking at an example. Using the running example from Fig. 2, we can look at the key steps in the CEGAR refinement. Let us assume we are only using the havoc strategy, i.e., the augmented CFA will look like shown in Fig. 4. Based on this CFA, an example for how a generic state-space exploration could look like is depicted in Fig. 5. In Fig. 5a we start at an abstract state with three components. The first one encodes the program location and is set to 2, since program location 2 is the initial progam location in the CFA. Component e_0 encodes the analysis-specific domain part of the abstract state, e.g., for predicate abstraction this could be a set of predicates. The last component is the strategy precision. It contains the base strategy (b) as well as the havoc strategy (h). From this state, the state-space exploration continues to program location 3, where the selection of strategies in the transition relation only allows us to proceed into the application of the havoc abstraction. From there, we eventually reach the error location.

This is where the CEGAR refinement operator is first called. Since the path formula to the error location is actually feasible, the wrapped refinement operator return the inputs unchanged, and our strategy refinement operator takes over. Here we discover that an overapproximating strategy was used on the path. We update the strategy precision of the second state (the one at program location 3) such that the havoc strategy cannot be chosen anymore. We then remove all successors of the pivot state from the set of reached abstract states (and the waitlist), add the modified state to the waitlist, and return both sets.

The resulting reachability graph will look like in Fig. 5b. From there, the state-space exploration can continue as shown in Fig. 5c. We again discover an error path, this time however the wrapped refinement operator can determine that this error path is infeasible. In case of a predicate abstraction, a predicate like $i < N$ would be discovered and added to the predicate precision of e'_1 at program location 3. All successors after location 3 are removed again and the wrapped refinement operator returns. Since there is no error state present anymore in the set of reached states, the strategy refinement operator returns its inputs unchanged. The state-space exploration then continues by adding a new abstract state for program location 4 and so on, as depicted in Fig. 5d.

Transformation into Source Code. We also provide functionality to convert the loop abstractions we found back into source code, such that our findings can be used and validated by others. For that, we provide two different mechanisms. The first is that whenever we are able to generate a proof using some loop-abstraction strategy, we generate a modified version of the input program where just the loops are changed to reflect the effect of the loop abstraction. The second mechanism is that we provide a way to analyze a C program such that for each loop in the program and each loop-abstraction strategy, we create a patch file for the program (in case the strategy is applicable) that —when applied— will apply the loop abstraction on the source-code level.

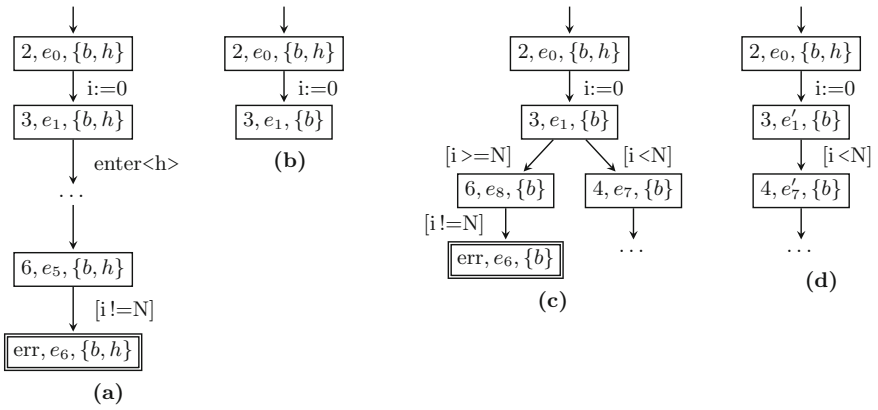


Fig. 5. Example for constructing a reachability graph of a program analysis on Fig. 4 using chained CEGAR refinements: **(a)** initial ARG until first refinement, **(b)** strategy precision updated after refinement (strategy h removed from precision), **(c)** state-space exploration on the original program continues, **(d)** exploration continues after a regular CEGAR refinement (e_1 replaced by e_1')

4 Evaluation

As a first step, we implemented the three loop-abstraction strategies that we described in Sect. 3.1 into the state-of-the-art verification framework CPAchecker: havoc abstraction (h), naive abstraction (n), and constant extrapolation (c). In addition, we also implemented so-called output abstraction (o) [15]. For the evaluation, we define the following (partial) order on which the function `select` will be based:

$$b \sqsubseteq o \sqsubseteq c \sqsubseteq n \sqsubseteq h \tag{3}$$

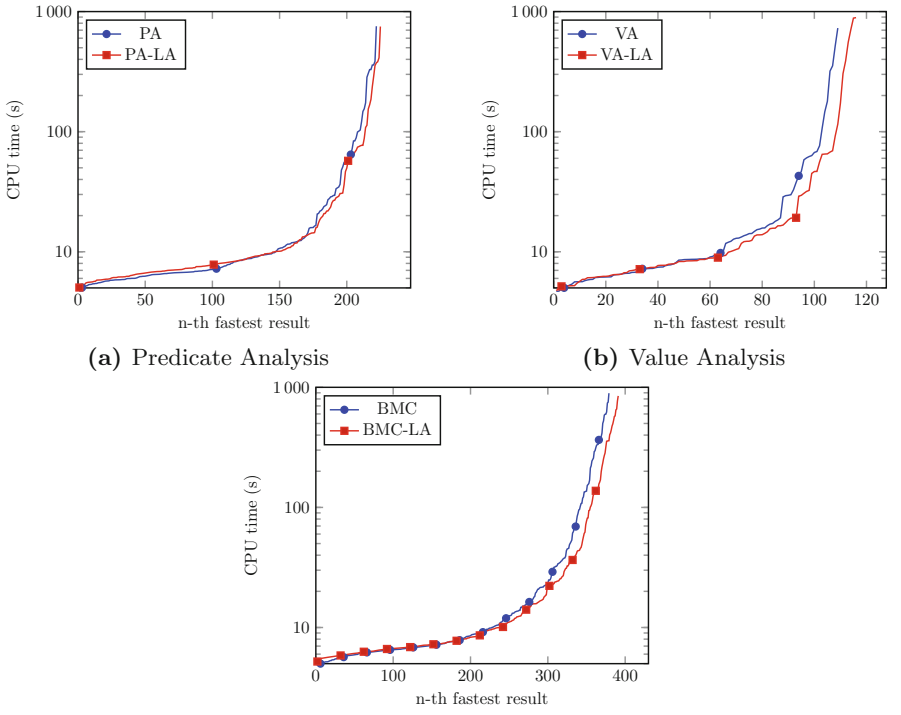
We are interested in answering the following research questions:

- **RQ1:** Can our CEGAR-style loop-abstraction scheme soundly improve a verifier like CPAchecker independently of the underlying analysis?
- **RQ2:** Are these abstractions also useful for other verifiers?

We conduct an experiment for each RQ in Sect. 4.2 to abtain answers.

4.1 Benchmark Environment

For conducting our evaluation, we use BenchExec to ensure reliable benchmarking [12]. All benchmarks are performed on machines with an Intel Xeon E5-1230 CPU (4 physical cores with 2 processing units each), 33GB of RAM, and running Ubuntu 20.04 as operating system. All benchmarks are executed with resources limited to 900 s of CPU time, 15 GB of memory, and 1 physical core (2 processing units).



(a) Predicate Analysis **(b)** Value Analysis

(c) BMC Analysis

Fig. 6. Quantile plots comparing performance of plain analyses with their versions that use loop-abstraction strategies; only correct results are considered

4.2 Experiments

For our experiments we use verification tasks taken from the SV-Benchmarks set of SV-COMP 2022 [3,4]. Here we selected only the 765 reachability tasks from the subcategory ReachSafety-Loops, as these cover a wide range of interesting loop constructs while at the same time only using a limited set of features of the programming language C, which allows us to focus on the algorithms instead of having to deal with lots of special cases.

RQ1. In a first experiment, we evaluate whether our approach can improve the overall results, and whether our new framework introduces significant overhead, for three analyses of CPAchecker: (1) predicate analysis (PA) [5] configured to use predicate abstraction [7,9,18], (2) value analysis (VA) [6,11], which is an extension of constant propagation [21], and (3) predicate analysis configured to work as bounded model checking (BMC) [13]. For improvements we will look at effectiveness as well as efficiency. By effectiveness we mean an increase in the number of solved verification tasks while at the same time preserving soundness of the results, i.e., no increase of the number of wrong proofs or wrong alarms. For efficiency we will take a look at how our approach affects the verification time of successfully verified tasks.

Table 1. Results for predicate abstraction (PA), value analysis (VA), and bounded model checking (BMC), without vs. with loop abstraction (LA)

	PA	PA-LA	VA	VA-LA	BMC	BMC-LA
Total	765	765	765	765	765	765
Total proofs	533	533	533	533	533	533
Correct proofs	164	163	33	35	235	248
Incorrect proofs	0	0	0	0	0	0
Total alarms	232	232	232	232	232	232
Correct alarms	58	62	76	81	144	144
Incorrect alarms	0	0	0	0	0	0

Table 2. Impact of loop abstractions on solving capabilities of the software verifiers UAutomizer (UA), Cbmc, and Symbiotic, without vs. with loop abstraction (LA) via generated abstracted programs

	UAutomizer	UAutomizer-la	Cbmc	Cbmc-la	Symbiotic	Symbiotic-la
Total	18	18	18	18	18	18
Total proofs	14	14	14	14	14	14
Correct proofs	12	13	0	13	12	13
Incorrect proofs	0	1	0	1	0	1
Total alarms	4	4	4	4	4	4
Correct alarms	0	3	1	3	1	3
Incorrect alarms	0	0	0	0	0	0

The quantile plots in Fig. 6 show that we are able to slightly improve the results for all analyses. Both effectiveness and efficiency is improved, and thus, there is no noticeable overhead. We use PA-LA, VA-LA, and BMC-LA to refer to the variants of the analyses that use our CEGAR-style loop-abstraction scheme. As expected, the overhead of applying loop abstraction in cases where this does not help with solving the verification task does not add a significant overhead to the verification time. Table 1 shows that our approach is also sound, i.e., it does not increase the number of incorrect results.

Another observation is that there are more proofs as well as property violations found this way. The latter is possible because constant extrapolation is a precise abstraction, meaning that a counterexample found using this strategy corresponds to a feasible error path in the program.

The experimental data so far suggests that if loop abstraction helps with verification, the verification will usually succeed very quickly. For all tasks where the verdict improves, the application of loop abstraction reduces the verification time from a timeout, i.e., more than 900 seconds, to less than 10 seconds. On closer inspection, we find a total of 18 verification tasks where the loop abstraction is essential in proving the program correct with the used analyses. When comparing the different analyses, the effect is most noticeable with bounded model checking, which is not surprising given the fact that BMC alone can not prove programs with unbounded loops. There are 6 tasks where predicate analysis improved, 5

tasks for value analysis, and 17 tasks for BMC.[4] Since our framework supports exporting the accelerated loops into the source code, we can use the 18 abstracted programs that improved CPAchecker's results in the next experiment, where we check whether these are also useful for other software verifiers.

RQ2. In the second experiment we take a look at whether our approach has the potential to improve the results of other state-of-the-art verification tools as well. In order to be able to do so without having to modify the existing tools, we take those programs where loop-abstraction strategies were able to improve the results for CPAchecker and automatically generate the abstracted programs that can then be fed to other verifiers. In our case, we use the three well-known verifiers Cbmc, Symbiotic, and UAutomizer.

The results of all three verifiers improve if loop abstraction is applied, as shown in Table 2. The table shows the results for the verifiers on the original verification tasks (columns without suffix LA) and on the abstracted programs (column with suffix LA). Note that this will not be the case in general, but for the selected verification tasks, we know that one of our implemented loop abstraction strategies is actually sufficient to prove the program correct. In general, if a loop abstraction over-approximates too much, the verifier will quickly find an error path, in which case we would execute the verifier on the original program. There is also one program for which our loop abstraction leads to a wrong proof, which is due to a bug in our translation back into source code.

The main observation here regarding our research question is that the results of all three verifiers can be improved by applying loop abstraction. We get the largest improvement for the bounded model checker Cbmc. This is not surprising and in line with the results from the bounded model checking with CPAchecker.

5 Conclusion

Loop abstraction is a technique for program verification that is currently not used by many of the state-of-the-art verification tools. In our experiments we have shown that mature verifiers can still benefit even from very simple loop abstractions. By adding more sophisticated loop-abstraction strategies in the future, we hope to achieve even better results that further improve the state-of-the-art. We make the loop abstractions that we implemented available to other tools by generating modified versions of the input programs, such that also other tools can benefit from loop abstractions in the future.

In this paper, we have also addressed the problem of how to select the right combination of loop abstractions for programs with multiple loops. Instead of deciding upfront which combination to choose, we use a novel approach based on CEGAR to automatically refine the loop abstractions as the analysis progresses. By using the control flow as interface for program analyses, we are able to apply our approach to a wide range of existing analyses and abstract domains, without additional implementation overhead.

[4] Detailed results at: https://www.sosy-lab.org/research/loop-abstraction/

Data-Availability Statement. The software and programs that we used for our experiments, including the generated programs with abstracted loops, are open source and available on our supplementary web page at https://www.sosy-lab.org/research/loop-abstraction/ and in the reproduction package at Zenodo [10].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

References

1. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141 (2019). https://doi.org/10.1109/ASE.2019.00121
2. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. PLDI. pp. 203–213. ACM (2001). https://doi.org/10.1145/378795.378846
3. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
4. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
5. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2017). https://doi.org/10.1007/s10817-017-9432-6
6. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
7. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer **9**(5–6), 505–525 (2007). https://doi.org/10.1007/s10009-007-0044-z
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13
9. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)
10. Beyer, D., Lingsch Rosenfeld, M., Spiessl, M.: Reproduction package for SEFM 2022 article 'A unifying approach for control-flow-based loop abstraction'. Zenodo (2022). https://doi.org/10.5281/zenodo.6793834
11. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
12. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
13. 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

14. 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
15. Darke, P., Chimdyalwar, B., Venkatesh, R., Shrotri, U., Metta, R.: Over-approximating loops to prove properties using bounded model checking. In: Proc. DATE. pp. 1407–1412. IEEE (2015). https://doi.org/10.7873/DATE.2015.0245
16. Darke, P., Khanzode, M., Nair, A., Shrotri, U., Venkatesh, R.: Precise analysis of large industry code. In: Proc. APSEC. pp. 306–309. IEEE (2012). https://doi.org/10.1109/APSEC.2012.97
17. Frohn, F.: A calculus for modular loop acceleration. In: Proc. TACAS (1). pp. 58–76. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_4
18. 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
19. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: Proc. POPL. pp. 529–540. ACM (2014). https://doi.org/10.1145/2535838.2535843
20. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15
21. Kildall, G.A.: A unified approach to global program optimization. In: Proc. POPL. pp. 194–206. ACM (1973). https://doi.org/10.1145/512927.512945
22. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Proc. TACAS (1). pp. 213–231. LNCS 10805, Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_12
23. Madhukar, K., Wachter, B., Kröning, D., Lewis, M., Srivas, M.K.: Accelerating invariant generation. In: Proc. FMCAD. pp. 105–111. IEEE (2015)
24. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6
25. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
26. Silverman, J., Kincaid, Z.: Loop summarization with rational vector addition systems. In: Proc. CAV, Part 2. pp. 97–115. LNCS 11562, Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_7

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)

LIV: Loop-Invariant Validation using Straight-Line Programs

Dirk Beyer
LMU Munich, Germany

Martin Spiessl
LMU Munich, Germany

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00214

***Abstract*—Validation of program invariants (a.k.a. correctness witnesses) is an established procedure in software verification. There are steady advances in verification of more and more complex software systems, but coming up with good loop invariants remains the central task of many verifiers. While it often requires large amounts of computation to construct safe and inductive invariants, they are more easy to automatically validate. We propose LIV, a new tool for loop-invariant validation, which makes it more practical to check if the invariant produced by a verifier is sufficient to establish an inductive safety proof. The main idea is to apply divide-and-conquer on the program level: We split the program into smaller, loop-free programs (a.k.a. straight-line programs) that form simpler verification tasks. Because the verification conditions are not encoded in logic syntax (such as SMT), but as programs in the language of the original program, any off-the-shelf verifier can be used to verify the generated straight-line programs. In case the validation fails, useful information can be extracted about which part of the proof failed (which straight-line programs are wrong). We show that our approach works by evaluating it on a suitable benchmark. Supplementary website: https://www.sosy-lab.org/research/liv/**

I. Introduction

Constructing and validating program invariants is the main task of many verification approaches. Since invariants can be proposed by imprecise approaches, it is imperative to *validate* whether each candidate invariant is indeed an inductive and safe program invariant. Verification witnesses have been proposed as an exchange format for program invariants [1], but there is a shortage of validators of correctness witnesses in software verification. It is extremely important to not only *confirm* invariants that can be used for the proof of correctness, but also to reliably *reject* invariants that are not helpful for constructing the correctness proof (they might be not inductive or not implying the safety property). Recently, an approach was proposed to split the verification task (C program and specification) to a set of C programs (with assertions inlined) such that the original program is correct if all generated C programs are correct [2]. We adopt and implement this approach in LIV, and explore its potential for the use case of validation of correctness witnesses.

Contributions. We contribute the following results:

- the open-source tool LIV, which is a witness validator that implements the approach of generating straight-line programs in the language of the original program,
- using off-the-shelf verifiers to establish correctness, and
- an evaluation on programs from the SV-Benchmarks repository [3], demonstrating the effectiveness of LIV.

II. Approach

For ease of presentation, we consider a basic programming language where a program is a sequence of statements (which can consist of other statements). Let Σ denote the set of all possible statements, then the set of all programs is denoted by Σ^*. For the empty program we will use the symbol ϵ. Each statement can either be an atomic statement (denoted by a), a compound statement (denoted by S), or a special statement. Special statements are statements that affect the control flow, such as the break and continue statements which are frequently used in loops and the goto statement[1]. Iteration statements like `while` C `do` B and branching statements like `if` C `then` S `else` T are compound statements.

We support the iteration statements to be annotated by a loop invariant γ, and write `while`$^\gamma$ C `do` B. If no invariant is annotated, we will implicitly assume the weakest possible invariant, that is, $true$. Location invariants can be added to every statement in the same manner, i.e., S^γ is a statement with a location invariant that holds whenever that statement is reached and before it is executed. A *straight-line program* is a program that does not contain any loop statement (and therefore also no loop-invariant annotations).

A Hoare triple consists of a precondition $\{P\}$ from the set $Pred$ of predicates, a program from Σ^*, and a post-condition $\{Q\}$ from $Pred$. A *straight-line Hoare triple* is a Hoare triple where the program does not contain any iteration statements.

Example. The program from Fig. 1a has the following statement structure:

$$s_0 \ \texttt{(if}\ C_1\ \texttt{(while}^\gamma\ C_2\ \texttt{do}\ B\texttt{)}\ \texttt{else}\ s_1\texttt{)}\ s_2$$

From the structure of the program, we can construct the following four straight-line Hoare triples that correspond to the straight-line programs shown in Figs. 1b, 1c, 1d, and 1e:

- $\{P\} s_0\, [C_1]\, \{\gamma\}$
- $\{\gamma \wedge C_2\} \mathrm{B} \{\gamma\}$
- $\{\gamma \wedge \neg C_2\} s_2 \{Q\}$
- $\{P\}\, [!C_1]\, s_1 s_2 \{Q\}$

The brackets around an expression, for example in $[C_1]$, indicate an assume statement. The predicates P and Q are both *true* in the example.

[1]We will not focus on goto statements in this paper, but plan to add support for this, which should be straight-forward.

DOI 10.1109/ASE56229.2023.00214

```
int x = nondet();
if (x>=0) {
  while (x>0) { // loop invariant: x>=0
    x--;
  }
} else x++;
y = x;
```

(a) Original program with a loop invariant

```
assume(1);
int x = nondet();
assume(x>=0);
assert(x>=0);
```

(b) SLP 1: loop invariant holds after initialization

```
int x = nondet();
assume(x>=0);
assume(x>0);
x--;
assert(x>=0);
```

(c) SLP 2: loop invariant is inductive

```
int x = nondet();
int y = nondet();
assume(x>=0);
assume(x<=0);
y = x;
assert(1);
```

(d) SLP 3: continuing after the loop

```
int x = nondet();
int y = nondet();
assume(1);
assume(x<0);
x++;
y = x;
assert(1);
```

(e) SLP 4: covering the else branch

Fig. 1: Example for splitting an original program into straight-line programs (SLPs) using a loop invariant; initializations in SLPs before the first assume are to produce valid C programs

Soundness. In general it is possible to prove the soundness of splitting procedures like the one we present here, which was done for VST-A [2], the main inspiration for LIV.

Our splitting procedure is implemented on top of the abstract syntax tree, and kept very simple on purpose. In addition, soundness bugs in our implementation can be expected to be discovered when applying LIV to the extensive benchmark set from the Competition of Software Verification, which covers many corner cases of the C language.

A. Tool Architecture

LIV is implemented in Python 3, using a modified version of *pycparser-ext*[2] as a frontend for parsing C programs. The splitting of the input program is done by traversing the abstract syntax tree of the input program. In addition, global variable and function definitions are collected and added to all generated straight-line programs. Verification of the generated straight-line programs is delegated to a backend verifier using CoVeriTeam [4]. The choice of the backend verifier can be configured (from a set of more than 45 verifiers for C programs [5]).

III. Evaluation

Our evaluation addresses the following research questions:

RQ 1 Can an (efficient) validator be constructed via splitting the original program into straight-line programs?

RQ 2 Can the validator give additional feedback to the user?

RQ 3 To which extent are the invariants provided by automatic software verifiers via correctness witnesses already enough to establish a complete, inductive proof?

[2]https://github.com/inducer/pycparserext

TABLE I: Results of running LIV on the benchmark set, using three off-the-shelf verifiers as backend

Verifier Backend	Correct (18 total)		Wrong (3 total)	
	Confirmed	Rejected	Confirmed	Rejected
CBMC	18	0	0	3
CPACHECKER	18	0	0	3
CPA-LIA	17	1	1	2

A. Experiment Setup

We conduct two experiments to show the usefulness of LIV, the first targeting RQ 1 and RQ 2, the second targeting RQ 3. For both experiments, we will look at the subset of verification tasks from the SV-Benchmarks repository[3] called *loop-zilu*, consisting of 22 C programs. For the first experiment, we will take invariants for the benchmark tasks from the ACSL-Benchmarks repository[4] where inductive loop invariants that should be sufficient to prove the assertion of the programs are annotated to the programs in various formats. For the second experiment, we use correctness witnesses that were produced by verifiers participating in SV-COMP 2022 [6].

We run our experiments on compute nodes with an Intel Xeon E3-1230 CPU. Each experimental run uses all 8 available processing units and is limited to use 15 GB of memory and 900 s of CPU time.

We configure LIV to run with different off-the-shelf verifiers as backend. For the first experiment we limit ourselves to CBMC [7] and CPACHECKER [8]. In addition, we use a special configuration of CPACHECKER which we refer to as CPA-LIA. This configuration uses linear integer arithmetic (LIA) as internal encoding, which is imprecise by design and allows us to observe cases in which the internal encoding makes a difference for validation.

For the second experiment, we only use CBMC as backend. Since the generated programs do not contain loops, a mature, bounded model checker like CBMC will allow us to quickly check the generated verification tasks while supporting a large subset of the C language.

In order to compare this to state-of-the art correctness validators, the second experiment also contains a comparison with CPACHECKER's correctness-witness validation, which is based on incremental k-induction, and which we refer to as CPA-KIND. Since the approach of LIV is more similar to 1-induction, we also compare with a modified version of CPAchecker's k-induction, where the induction bound k is fixed to 1, hence we refer to it as CPA-1IND.

B. Evaluation Results

RQ 1. Table I shows the results for the first experiment. For our benchmark set, all correct witnesses are confirmed by the off-the-shelf verifiers CBMC and CPACHECKER. These also correctly reject three witnesses for which the invariant is actually not sufficient. One such example of an inductive invariant that is not

[3]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks
[4]https://gitlab.com/sosy-lab/research/data/acsl-benchmarks

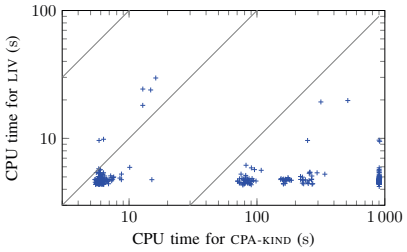
```c
int main() {
    int k = __VERIFIER_nondet_int();
    int j = __VERIFIER_nondet_int();
    int n = __VERIFIER_nondet_int();

    if (!(n>=1 && k>=n && j==0)) return 0;
    //@ loop invariant j <= n && n <= k + j;
    while (j<=n-1) {
      j++;
      k--;
    }
    //@ assert k >= 0;
    __VERIFIER_assert(k>=0);
    return 0;
}
```

(a) **benchmark04_conjunctive.c**: Inductive but unsafe invariant (invariant holds after initialization, is inductive, but does not imply the assertion)

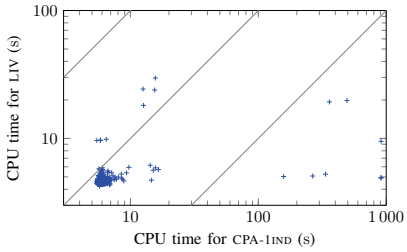
```c
int main() {
    int i = __VERIFIER_nondet_int();
    int c = __VERIFIER_nondet_int();
    if (!(c==0 && i==0)) return 0;
    //@ loop invariant 2 * c == (i-1) * i
        ↪ && 0 <= i && i <= 100;
    while (i<100) {
      c = c+i;
      i = i+1;
      if (i<=0) break;
    }
    //@ assert c >= 0;
    __VERIFIER_assert(c>=0);
    return 0;
}
```

(b) **benchmark10_conjunctive.c**: Invariant not inductive (integer variable `c` can have negative values due to overflow)

Fig. 2: Example for benchmarks with insufficient invariants



(a) Comparison with CPA-KIND



(b) Comparison with CPA-1IND

Fig. 3: Scatter plots comparing LIV with CPACHECKER-based correctness-witness validation; all results are shown, independent of the verification result

sufficient to prove the assertion is shown in Fig. 2a. CPA-LIA misses rejecting one incorrect witness and instead rejects one witness that is actually correct. The incorrect invariant that is accepted by CPA-LIA is shown in Fig. 2b.

Regarding efficiency, Fig. 3a shows that the execution of LIV is also efficient, i.e., it finishes after a few seconds in the majority of cases and never runs into a timeout for the given benchmark set. This is also mostly true for CPA-1IND (Fig. 3b); there are only a few outliers and timeouts caused by the employed SMT solver. The fact that CPA-1IND confirms more witnesses than LIV is due to the fact that CPA-1IND unrolls the loop body one additional time (to encode assertions inside the loop body) before checking the assertion after the loop. CPA-KIND completely unrolls the loops in several programs that have a finite loop bound, which is why it confirms significantly more witnesses.

In sum, RQ 1 can be answered positively.

RQ 2. Initially we expected all the witnesses from the benchmark set to be confirmed. Surprisingly, upon inspection of the witnesses that LIV rejected, we indeed confirmed that some invariants are not strong enough to establish the specified safety property. We show two such examples in Fig. 2.

Upon failure, existing validators would at most output information about whether a specific invariant was confirmed or not. LIV can give more fine-grained reports, as we can distinguish between whether the invariant holds after the initialization, whether it is inductive, or whether the invariant does not ensure the safety property (assertion after the loop). The two programs in Figs. 2a and 2b are examples in which LIV tells us why the proof of correctness cannot be established.

In sum, LIV's feedback is for each proof step by construction.

RQ 3. The results for the second experiment are shown in Table II. We show only the results for verifiers that created non-trivial correctness witnesses in SV-COMP 2022, i.e., witnesses that actually contain at least one syntactically valid invariant.

We can observe that a significant fraction of the invariants reported by the verifiers are actually sufficient for an inductive proof by LIV. Upon closer inspection of the output of the two CPACHECKER-based variants, we can observe that it often happens that they ignore the provided invariants, leading to confirmed results even if the provided invariants are not sufficient. Also, neither of the CPACHECKER-based approaches rejected any witness due to a wrong invariant.

For RQ3, while there are significantly many witnesses that help proving correctness, we also report that there is still room for improvement in the invariants provided from the verifiers.

IV. Related Work

Witness Validation. There are only few approaches for validating correctness witnesses [9, 10]. The closest to our approach is METAVAL [9]. METAVAL encodes additional proof goals for the invariants provided via the witness into a C program, but does not split programs up into multiple, simple sub-programs. The C program is constructed as automaton product of the

TABLE II: Results for LIV and CPAchecker-based witness validation on the SV-COMP witnesses from each verifier

Verifier	# Tasks		LIV				CPA-kind		CPA-1ind	
	total	non-trivial	confirmed	rejected	unknown	error	confirmed	unknown	confirmed	unknown
2LS	13	12	6	7	0	0	11	2	7	6
CBMC	7	7	1	5	1	0	7	0	3	4
CVT-AlgoSel	16	11	2	13	1	0	9	7	5	11
CVT-ParPort	19	5	4	15	0	0	14	5	9	10
CPAchecker	21	6	5	14	0	2	15	6	10	11
Graves	22	9	5	14	2	1	12	10	10	12
PeSCo	21	16	11	5	1	4	15	6	15	6
UAutomizer	22	22	9	12	1	0	11	11	7	15
UKojak	21	21	10	10	1	0	10	11	7	14
UTaipan	22	22	6	16	0	0	11	11	7	15

witness automaton and the CFA of the original program, and as such it is not immediately clear whether this construction is sound, and less likely to be faster to validate. In case the validation fails, it is not clear to the user where exactly the proof goes wrong, and loops are still present.

Verification-Condition Generation. Our design can also be seen as a variation of deductive verification, where verification conditions are generated from the (user-)provided invariants, which are then typically handed off to a backend solver. For example, Dafny [11] translates to Boogie [12, 13] as an intermediate representation. Other examples include VeriFast [14] and Frama-C [15]. The automatic verifier Korn [16] translates to constrained Horn clauses. While for existing deductive and automatic verifiers, the verification conditions usually use a very specific representation that is bound to the particular backend and cannot easily be reused, LIV uses the original programming language to encode the verification conditions and off-the-shelf verifiers as backend to solve them. Our approach is closely motivated by VST-A [2, 17], which is an annotation verifier for C programs. The main difference is that our transformation is purely AST-based, while for VST-A the actual splitting is defined over the CFA of the program. In general, algorithms for verification-condition generation are tool-specific and not described in detail in literature, with few exceptions [18].

V. Conclusion

There is a demand for invariant-validation tools in order to ensure the correctness of the results of verification tools. LIV is a new validator for correctness witnesses. The unique feature of this tool is that it reads the invariants from correctness witnesses, and reduces the validation problem to the verification of a set of straight-line programs. This enables the application of arbitrary off-the-shelf verification tools for C programs, including bounded verifiers, for establishing a proof of correctness.

Acknowledgement. This work was inspired by a discussion with Andrew Appel at the Isaac Newton Institute, Cambridge, in July 2022.

Data-Availability Statement. The source code of LIV is available at https://gitlab.com/sosy-lab/software/liv and a reproduction package [19] includes the tool and experimental data. The results of our experiments are available in interactive BenchExec tables at https://www.sosy-lab.org/research/liv/.

Funding Statement. This project was funded in part by the Deutsche Forschungsgem. (DFG) – 378803395 (ConVeY).

References

[1] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). doi:10.1145/3477579

[2] Zhou, L.: Foundationally sound annotation verifier via control flow splitting. In: Proc. SPLASH. pp. 69–71. ACM (2022). doi:10.1145/3563768.3563956

[3] Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). doi:10.5281/zenodo.5831003

[4] Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). doi:10.1007/978-3-030-99524-9_31

[5] Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_29

[6] Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). doi:10.5281/zenodo.5838498

[7] Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). doi:10.1007/978-3-540-24730-2_15

[8] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). doi:10.1007/978-3-642-22110-1_16

[9] Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). doi:10.1007/978-3-030-53291-8_10

[10] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). doi:10.1145/2950290.2950351

[11] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proc. LPAR. pp. 348–370. LNCS 6355, Springer (2010). doi:10.1007/978-3-642-17511-4_20

[12] DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)

[13] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proc. FMCO. pp. 364–387. LNCS 4111, Springer (2005). doi:10.1007/11804192_17

[14] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. NFM. pp. 41–55. LNCS 6617, Springer (2011). doi:10.1007/978-3-642-20398-5_4

[15] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). doi:10.1007/978-3-642-33826-7_16

[16] Ernst, G.: Korn: Horn clause based verification of C programs (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_36

[17] Wang, Q., Cao, Q.: VST-A: A foundationally sound annotation verifier. arXiv/CoRR **1909**(00097) (August 2019). doi:10.48550/arXiv.1909.00097

[18] Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. Comput. J. **38**(2), 131–141 (1995). doi:10.1093/comjnl/38.2.131

[19] Beyer, D., Spiessl, M.: Reproduction package for ASE 2023 article 'LIV: Invariant validation using straight-line programs'. Zenodo (2023). doi:10.5281/zenodo.8289101

Software Verification Witnesses 2.0

Paulína Ayaziová [1], Dirk Beyer [2]✉, Marian Lingsch-Rosenfeld [2], Martin Spiessl [2], and Jan Strejček [1]

[1] Masaryk University, Brno, Czech Republic
{xayaziov,strejcek}@fi.muni.cz
[2] LMU Munich, Munich, Germany
{dirk.beyer,marian.lingsch,spiessl}@sosy.ifi.lmu.de

Abstract. Verification witnesses are now widely accepted objects used not only to confirm or refute verification results, but also for general exchange of information among various tools for program verification. The original format for witnesses is based on GraphML, and it has some known issues including a semantics based on control-flow automata, limited tool support of some format features, and a large size of witness files. This paper presents version 2.0 of the witness format, which is based on YAML and overcomes the above-mentioned issues. We describe the new format, provide an experimental comparison of various aspects of the original and the new witness format showing that both witness formats perform similarly, and report on its adoption in the community.

Keywords: Verification Witness · Software Verification · Validation · Exchange Format · Invariant · Counterexample

1 Introduction

Software verification is a process that detects bugs in computer programs or proves their absence. Unfortunately, software verifiers can also contain bugs and their verdicts can thus be incorrect. To increase the reliability of the verification process, starting eight years ago, software verifiers have accompanied their verification results with witnesses that justify the verdict and can be independently analyzed by witness validators developed by various teams and based on different techniques.

The first generic format for witnesses of verification results [1] was introduced in 2015. It supported only *violation witnesses* (also called *counterexamples*) produced when a verifier reports that a given program violates a considered safety specification. In 2016, the format was extended to accommodate also witnesses for the cases when a verifier decides that a given program satisfies a given specification [2]. Such witnesses are called *correctness witnesses*, and they should contain invariants that help to prove that the program is correct. The format was soon adopted by the verification community and by the *Competition on Software Verification (SV-COMP)* [3], which led to fast adoption of the format by many verification tools and to the development of numerous witness validators.

T. Neele and A. Wijs (Eds.): SPIN 2024, LNCS 14624, pp. 184–203, 2025.
https://doi.org/10.1007/978-3-031-66149-5_11

The overview of existing validators can be found in a recent survey [4]. Since 2023, SV-COMP has a new track on witness validation [5].

While the format was originally intended for validation of verification results, some witness validators can also refute a witness [4, 5]. The format soon found also some applications that were not intended at the time of its development. In particular, it is used to exchange information between different verifiers in the context of cooperative verification [6, 7], as a way to provide feedback to a software developer [8, 9], or as a way to combine automatic and interactive verifiers [10]. In 2022, the authors of the format published a paper [11] with its detailed description and with an extensive experimental study on its applications.

Despite the indisputable success of the format, it has also some weaknesses. The format is based on GraphML [12] and witnesses have the form of automata, which makes them easy to visualize, but also lengthy and unsuitable for reading in their textual form. More importantly, the semantics of the format is formally defined over programs represented by *control-flow automata (CFA)*. Unfortunately, there is no standardized translation of programs written in common programming languages like C or Java to CFA. As a result, the semantics of the format over programs in standard languages has some ambiguities. The SV-COMP community even found a part of the semantics related to implicit loop edges as inappropriate and decided to change it. Another issue of the original witness format is connected to the high number of features it provides. For example, if an invariant or an assumption uses variables that appear in different functions or scopes, the format allows to specify the scope for their interpretation. Another example is that the location of some witness event can be specified very loosely by an interval of lines. Practical experience shows that some of these features are not used in any witness generated by verifiers and, what is more alarming, unsupported or even ignored by witness validators. In fact, there is probably no witness validator fully implementing the format. This can lead to the situation in which a valid verification witness employing some less frequent feature is not confirmed or even refuted, or an invalid witness is confirmed.

This paper presents a new generation of the witness format that avoids the mentioned weaknesses. In particular, we use a concise format that is based on YAML, which makes the witnesses shorter in general. Further, the format provides only features that were really used by verification witnesses in the original format. As the format is significantly simpler, it is easy to fully support it by validators. Finally, the semantics of the format is formulated over programming languages using terms and concepts from their standards.

The format itself is described in Sect. 2 and referred to as version 2.0. In its current state, the format supports only sequential programs written in C and basic program properties, namely unreachability of a given error function, unreachability of signed integer overflow, and unreachability of invalid pointer dereference and deallocation. We have adopted two verification tools, namely CPAchecker [13] and Symbiotic [14], to produce verification witnesses in the new format. We have also developed two witness validators, namely CPAchecker [11] (as an extension of the exiting tool) and Witch3 (new validator based on the

concept of Witch3 [15]) to validate witnesses in this format. Using these tools and other tools from the competition, Sect. 3 evaluates the impact of the new format on the witnesses and their validation. Sect. 4 summarizes the differences between the original and the new format and shows the current adoption of the new format by verification and witness validation tools.

Contributions. In this paper, we contribute:

- a new generation of the format for verification witnesses that solves most problems that were present in the previous format,
- a preliminary evaluation of the impact of the new format on the effectivity and performance of witness validation, and
- an overview of a few measures that characterize the new witnesses.

Related Work. Our work certainly stands on the shoulders of the original format for verification witnesses [1, 2], but we claim to provide a substantial improvement over the original format by addressing its weaknesses (see Sect. 4). Witnesses are used ubiquitously in areas where algorithms have a high computational complexity. For example, witnesses are used for certifying graph algorithms [16]. Turing used assertions [17] already and argued that one should justify the correctness of programs. In the area of logic solvers, witnesses for the results are of essence for competitions, and important competitions require witnesses and their validation. For example, the *Termination Competition (termCOMP)* [18] uses the format CPF [19], the competition of SAT solvers [20] uses the DRAT format [21] together with the validator DRAT-trim [22], and the competition on SMT solving verifies models with Dolmen [23].

Witnesses are not only important to certify the correctness of a solver's answer, it is also important for the goal of explainability: The *true* / *false* answers alone are not as valuable compared to also providing the reasons to understand the answer. For example, witnesses can be used to derive test cases [9] and to aid debugging with visualizations [8]. Execution reports [24] help organize the analysis results, and the format SARIF [25] is used by static analyzers to represent results.

2 Witness Format 2.0

The witness format 2.0 is an extension of the YAML format, version 1.2. Individual verification witnesses are represented by *entries*. Each entry has three key-value pairs. The key `entry_type` has the value `invariant_set` or `violation_sequence` corresponding to the type of the witness: a correctness witness is represented by one or more entries of type `invariant_set`, while a violation witness is represented by a single entry of type `violation_sequence`. Further, the key `metadata` refers to a mapping that describes mainly the context of the witness: the format version used by the entry, the unique identifier of the entry, the creation time of the entry, the tool that produced the entry, and the verification task the witness relates to. Finally, the value of the key `content` represents the semantical content

Table 1: Structure of entries common for violation and correctness witnesses; some nodes are nested; optional items are marked with *; the term *scalar* in YAML refers also to strings

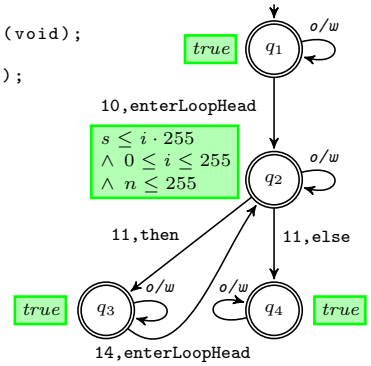
Key	Value	Description
`entry_type`	`invariant_set`	the entry type of a correctness witness
	`violation_sequence`	the entry type of a violation witness
`metadata`	mapping	the context of the witness; see below
`content`	sequence	the witness content; see Tables 2 and 3
content of `metadata`		
`format_version`	`2.0`	the used version of the format
`uuid`	scalar	a unique identifier of the entry; it uses the UUID format defined in RFC4122
`creation_time`	scalar	the date and time of the entry creation; it uses the format given by ISO 8601
`producer`	mapping	the tool that produced the entry; see below
`task`	mapping	the verification task to which the entry is related; see below
content of `producer`		
`name`	scalar	the name of the tool
`version`	scalar	the version of the tool
`configuration`*	scalar	the configuration in which the tool ran
`command_line`*	scalar	the command line with which the tool ran; it should be a bash-compliant command
`description`*	scalar	any additional information
content of `task`		
`input_files`	sequence	the list of files given as input to the verifier, e.g. `["path/f1.c", "path/f2.c"]`
`input_file_hashes`	mapping	SHA-256 hashes of all files in `input_files`, e.g. `{"path/f1.c": 511..., "path/f2.c": f70...}`
`specification`	scalar	the property considered by the verifier; it uses the SV-COMP format given at https://sv-comp.sosy-lab.org/2024/rules.php
`data_model`	`ILP32` or `LP64`	the data model considered for the task
`language`	`C`	the programming language of the input files; the format currently supports only C

of the entry. The key-value pairs are presented in a structured way in Table 1. The table also presents the key-value pairs of the nested mapping `metadata` and its nested mappings `producer` and `task`. We describe the possible values of the

188 P. Ayaziová, D. Beyer, M. Lingsch-Rosenfeld, M. Spiessl, and J. Strejček

```c
void reach_error(){}
extern unsigned char __VERIFIER_nondet_uchar(void);
int main() {
  unsigned char n = __VERIFIER_nondet_uchar();
  if (n == 0) {
    return 0;
  }
  unsigned char v = 0;
  unsigned int  s = 0;
  unsigned int  i = 0;
  while (i < n) {
    v = __VERIFIER_nondet_uchar();
    s += v;
    ++i;
  }
  if (s < v) {
    reach_error();
    return 1;
  }
  if (s > 65025) {
    reach_error();
    return 1;
  }
  return 0;
}
```

Specification:
`G ! call(reach_error())`,
i.e., all calls of `reach_error()`
are unreachable




```yaml
- entry_type: invariant_set
  metadata: <...>
  content:
    - invariant:
        type: loop_invariant
        location:
          file_name: "inv-a.c"
          line: 11
          column: 1
          function: main
        value: "s <= i*255 && 0 <=
        ↪ i && i <= 255 && n <= 255"
        format: c_expression
```

Fig. 1: Example C program inv-a.c taken from [11] (top left) satisfying the given specification (bottom left) and equivalent correctness witnesses in format 1.0 (top right, visualized as automaton) and format 2.0 (bottom right), with a single nontrivial invariant

key `content` in the following subsections separately for correctness witnesses and violation witnesses as they are conceptually different.

2.1 Correctness Witnesses

Correctness witnesses provide invariants that should help to prove the program correct. In the old format (1.0), invariants are tied to automata nodes and these nodes can correspond to multiple program locations and various moments of program executions. The new format (2.0) simply assigns invariants to program locations. Figure 1 provides an example of a correctness witness in the old format and in the new format.

Syntax. In entries of type `invariant_set` which represent a correctness witness, the key `content` contains a sequence of zero or more *invariants*. An `invariant` is a mapping with the following four keys.

`type` has the value `loop_invariant` if the invariant is assigned to a loop head and the value `location_invariant` if it is assigned to another location.

Table 2: Structure of the `content` part of entries representing correctness witnesses; optional items are marked with *

Key	Value	Description
`content`	sequence	a sequence of one or more `invariant` elements
	description of `invariant`	
`invariant`	mapping	a basic building block of correctness witnesses; see below
	content of `invariant`	
`type`	`loop_invariant`	the invariant type for iteration statements
	`location_invariant`	the invariant type for arbitrary statements
`location`	mapping	the location of the invariant; see below
`format`	`c_expression`	the invariant is a C expression
`value`	scalar	the actual invariant
	content of `location`	
`file_name`	scalar	the file of the location
`line`	scalar	the line number of the location
`column`*	scalar	the column of the location
`function`*	scalar	the name of the function containing the location

`location` of a `loop_invariant` must point to the first character of a keyword at the beginning of a loop (i.e., `for`, `while`, or `do`). The `location` of a `location_invariant` must point to the first character of a statement or a declaration that is within a compound statement.

`format` has the value `c_expression` as the format currently supports only invariants that are C expressions.

`value` holds the actual invariant string (e.g., `"s <= i*255 && i > 0"`), which is a side-effect-free C expression over variables in the scope where the invariant is placed.

The `location` is a mapping with mandatory keys `file_name` that holds the name of the file and `line` representing the line number (the first line has the number 1). Additionally, there are two optional keys called `column` and `function`. The key `column` specifies the column number of the location (value 1 is the position of the first character on the `line`). If the column is not given, then it is interpreted as the leftmost suitable position on the line, where suitability is given by `type` and the restrictions given above. The key `function` provides the name of the function containing the location. Technically, this information is superfluous as it is determined by the `file_name`, `line`, and `column`. It is therefore not intended for any algorithmic processing of the witness, but only to improve human readability of the witness.

The structure of `content` and its nested items are summarized in Table 2.

Semantics. The correctness witness is *valid* if it fulfills the following requirements.

- Each `loop_invariant` must always hold immediately before evaluating the condition of the corresponding loop.
- Each `location_invariant` must always hold immediately before evaluating the corresponding statement or declaration.
- The specification must be satisfied for all program executions.
- No invariant evaluation causes undefined behavior and no undefined behavior occurs during any execution of the program.

Note that the order of invariants in an `invariant_set` or their division into several entries of type `invariant_set` is not important. The semantics also reveals the difference between the two types of invariants: if we replace `loop_invariant` with `location_invariant`, then the invariant has to hold only before the loop is executed, but not after each loop iteration.

2.2 Violation Witnesses

A violation witness should describe a program execution violating the considered property. For brevity, the violating execution is described loosely and the witness thus represents a set of such executions. In the old format (1.0), a violation witness is an automaton with edges prescribing consecutive restrictions on program executions. The automaton can contain various branches and loops. In the new format (2.0), a violation witness is a sequence of *waypoints* that have to be passed by the executions. To make the witness validation more efficient, the format also allows specifying waypoints that have to be avoided. Figure 2 provides an example of a violation witness in the old format and in the new format.

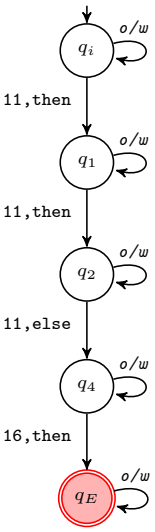
Syntax. The basic building blocks of violation witnesses in the new format are waypoints. Technically, a `waypoint` is a mapping with four keys, namely `type`, `location`, `constraint`, and `action`. The values of the first three keys specify a requirement on a program execution to pass a waypoint: `type` describes the type of the requirement, `location` ties the requirement to some program location, and `constraint` gives the requirement itself. The key `action` then states whether the executions represented by the witness should pass through the waypoint (value `follow`) or avoid it (value `avoid`). The format currently supports five possible values of `type` with the following meanings:

`assumption` The `location` has to point to the first character of a statement or a declaration within a compound statement. A requirement of this type says that a given constraint holds before evaluating the pointed statement or declaration. The `constraint` is a mapping with two keys: `format` specifies the language of the assumption and `value` contains a side-effect-free assumption over variables in the current scope. The value of `format` is `c_expression` as C expressions are the only assumptions currently supported. In the future, we plan to support also assumptions in ACSL [26].

`branching` A requirement of this type says that a given branching is evaluated in a given way. The `location` points to the first character of a branching keyword like `if`, `while`, `switch`, or to the character `?` in the ternary operator (`?:`).

```c
void reach_error(){}
extern unsigned char
 ↪ __VERIFIER_nondet_uchar(void);
int main() {
  unsigned char n = __VERIFIER_nondet_uchar();
  if (n == 0) {
    return 0;
  }
  unsigned char v = 0;
  unsigned char s = 0;
  unsigned int  i = 0;
  while (i < n) {
    v = __VERIFIER_nondet_uchar();
    s += v;
    ++i;
  }
  if (s < v) {
    reach_error();
    return 1;
  }
  if (s > 65025) {
    reach_error();
    return 1;
  }
  return 0;
}
```

Specification:
`G ! call(reach_error())`,
i.e., all calls of `reach_error()`
are unreachable




```yaml
- entry_type:
 ↪ violation_sequence
  metadata: <...>
  content:
    - segment:
      - waypoint:
          action: follow
          type: branching
          location:
            file_name:
             ↪ "inv-b.c"
            line: 11
          constraint:
            value: true
    - segment:
      - waypoint:
          action: follow
          type: branching
          location:
            file_name:
             ↪ "inv-b.c"
            line: 11
          constraint:
            value: true
    - segment:
      - waypoint:
          action: follow
          type: branching
          location:
            file_name:
             ↪ "inv-b.c"
            line: 11
          constraint:
            value: false
    - segment:
      - waypoint:
          action: follow
          type: target
          location:
            file_name:
             ↪ "inv-b.c"
            line: 17
```

Fig. 2: Example C program inv-b.c taken from [11] (top left) violating the given specification (bottom left) and similar violation witnesses in format 1.0 (middle, visualized as automaton) and format 2.0 (right)

The `constraint` is then a mapping with only one key `value`. For binary branchings, `value` can be either `true` or `false` saying whether the true branch is used or not. For the keyword `switch`, `value` can be an integer constant or `default`. The integer constant specifies the value of the controlling expression of the `switch` statement. The value `default` says that the value of this expression does not match any case of the `switch` with the exception of the `default` case (if it is present).

`function_enter` The `location` points to the right parenthesis after the function arguments of a function call. The requirement says that the called function is entered. The key `constraint` has to be omitted in this case.

`function_return` Such a requirement says that a given function call has been evaluated and the returned value satisfies a given constraint. The `location` points to the right parenthesis after the function arguments at the function call. The `constraint` is a mapping with keys `format` and `value`. We currently support only ACSL expressions of the form `\result <op> <const_expression>`, where `<op>` is one of `==`, `!=`, `<=`, `<`, `>`, `>=` and `<const_expression>` is a constant expression. The value of `format` has to be `acsl_expression`.

`target` This type of requirement can be used only with action `follow` and it marks the program location where the property is violated. More precisely, the `location` points at the first character of the statement or full expression whose evaluation is sequenced directly before the violation occurs, i.e., there is no other evaluation sequenced before the violation and after the sequence point associated with the `location`. This also implies that it can point to a function call only if it calls a function of the C standard library that violates the property or if the function call itself is the property violation. The key `constraint` has to be omitted.

Waypoints are organized into *segments*. Each `segment` is a sequence of zero or more waypoints with action `avoid` and exactly one waypoint with action `follow` at the end. A segment is called *final* if it ends with the `target` waypoint and it is called *normal* otherwise.

Finally, we can describe the `content` part of `violation_sequence` entries which represent violation witnesses. The value of `content` is a sequence of zero or more normal segments and exactly one final segment at the end. The structure of `content` and its nested items are summarized in Table 3.

Semantics. Each violation witness represents a set of some program executions violating the specified property. The witness is considered to be *valid* if the set is nonempty.

Let us consider a violation witness with $n \geq 1$ segments. An execution is represented by this witness, if the execution can be divided into n parts such that, for each $1 \leq i \leq n$, the i-th part matches the corresponding segment of the witness. An execution part matches a normal segment if

- it does not pass any avoid waypoint of the segment,
- it ends in the moment when the sequence point corresponding to the follow waypoint of the segment is entered for the first time in the execution part, and
- the follow waypoint is passed in this moment.

The final execution part matches the final segment if

- it does not pass any avoid waypoint of the segment and
- it violates the considered property during execution of the statement identified by the target waypoint.

Moreover, the execution must not contain any instruction that causes undefined behavior. An exception to this are witnesses of undefined behavior, in

Table 3: Structure of the `content` part of entries representing violation witnesses

Key	Value	Description
`content`	sequence	a sequence of zero or more normal `segment` elements and one final `segment` at the end
	description of `segment`	
`segment`	sequence	a sequence of zero or more `waypoint` elements with `action: avoid` and one `waypoint` with `action: follow` at the end; the final segment ends by `waypoint` with `type: target`
	description of `waypoint`	
`waypoint`	mapping	a basic building block of violation witnesses
	content of `waypoint`	
`action`	`follow`	the waypoint should be passed through
	`avoid`	the waypoint should be avoided
`type`	`assumption`	restriction on variable values given by an expression
	`branching`	restriction specifying the result of a branching
	`function_enter`	restriction saying that a function is entered
	`function_return`	restriction on the result of a function call
	`target`	identification of a location of the property violation
`location`	mapping	the location of the waypoint; see Table 2
`constraint`	mapping	the constraint of the waypoint; not allowed with `type: function_enter` and `type: target`
	content of `constraint`	
`format`	`c_expression`	for `type: assumption`, constraints are C expressions
	`acsl_expressions`	for `type: function_return`, constraints are specific ACSL expressions; not allowed for other `type` values
`value`	scalar	the actual constraint

which case the only instruction that causes undefined behavior must be the one represented by the target waypoint.

In each execution part, only the waypoints of the corresponding segment are evaluated. An `assumption` waypoint is evaluated at the sequence point immediately before the waypoint location. The evaluation must not lead to undefined behavior; otherwise the witness is incorrect. The waypoint is passed if the given constraint evaluates to true. A `branching` waypoint is evaluated at the sequence point immediately after evaluation of the controlling expression of the corresponding branching statement. The waypoint is passed if the resulting value of the controlling expression corresponds to the given constraint. A `function_enter` waypoint is evaluated at the sequence point immediately after evaluation of all arguments of the function call. The waypoint is passed without any additional constraint. A `function_return` waypoint is evaluated immediately after evalua-

tion of the corresponding function call. The waypoint is passed if the returned value satisfies the given constraint.

3 Evaluation

This section presents experiments with validation of verification results using both formats (1.0 and 2.0) to answer the following research questions:

- **RQ 1:** How does the performance of the validation of the new witness format compare to the old witness format?
- **RQ 2:** Does the new format improve attributes related to readability when compared to the old format?

In the experiments, the following tools were used.

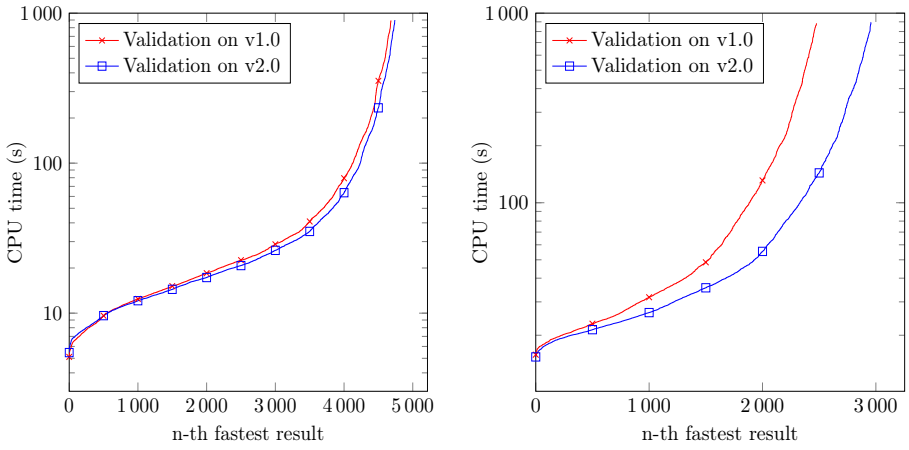
- CPAchecker [13, 27] is a verifier and witness validator that can produce and validate both correctness and violation witnesses in both formats. The experiments are based on version `0af0e41240`.
- Symbiotic [28] is a verifier that can produce violation witnesses in both formats. We use version `9c278f9`.
- Symbiotic-Witch2 [15] is a witness validator for violation witnesses in the old format (1.0). The experiments are based on version `svcomp24`.
- Witch3 [29] is a new witness validator based on similar principles as Symbiotic-Witch2, but designed for violation witnesses in format 2.0. The tool is made of Symbiotic in version `b011ec9` and Witch-Klee in version `6dabb94`.
- UAutomizer [30] is a verifier and witness validator that can produce both correctness and violation witnesses in both formats and validate correctness witnesses in both formats and violation witnesses only in format 1.0. We use version `0.2.4-?-8430d5a-m` and version `0.2.4-dev-0e0057c` for validation of YAML and GraphML correctness witnesses respectively.

Note that the support of the new witness format in all mentioned tools except UAutomizer has been implemented by authors of this paper.

For the experiments, we used all verification tasks of SV-COMP 2024 where the property to be verified is *unreachabilty of error function*, i.e., the specification used in Figs. 1 and 2. We did not use the witnesses produced during the competition [31], but rather based our experiments on fresh witnesses produced by the latest versions of Symbiotic and CPAchecker; the results are much better compared to the results from SV-COMP 2024 [32].

Benchmark Environment. For conducting our evaluation, we use BenchExec to ensure reliable benchmarking [33]. All benchmarks are performed on machines with an Intel Xeon E5-1230 CPU (4 physical cores with 2 processing units each), 33 GB of RAM, and running Ubuntu 22.04 as operating system. Each verification and witness validation task is executed with resource limits used in SV-COMP, i.e., 900 s of CPU time[3], 15 GB of memory, and 1 physical core (2 processing units).

[3] Except violation witness validation, where the convention is to use 90 s of CPU time.



(a) Validation using CPAchecker **(b)** Validation using UAutomizer

Fig. 3: Correctness witnesses produced by CPAchecker: Quantile plots for the time taken for validation of the old and new witnesses for two different validators

3.1 Evaluation Results for RQ 1 (Validation Performance)

One of the most important questions is whether the validation using the new format is as effective and efficient as with the old witness format.

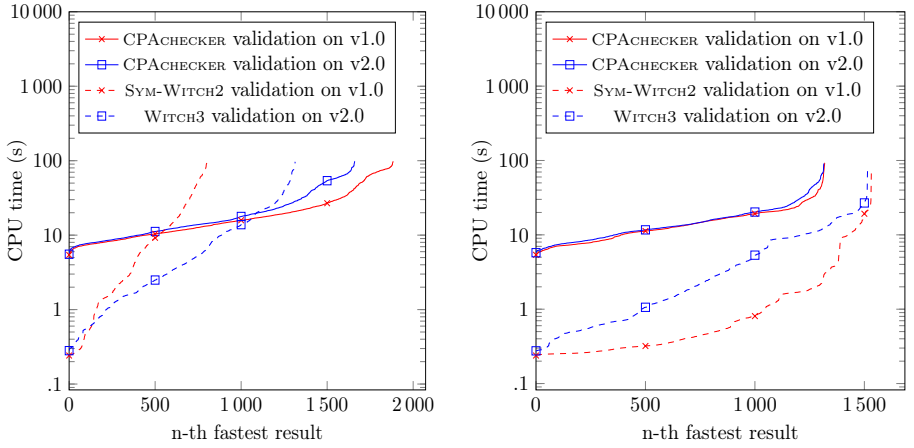
Correctness Witnesses. CPAchecker can generate correctness witnesses in both formats. The witnesses from CPAchecker were then validated by CPAchecker and UAutomizer. This allows for a direct comparison as shown in Fig. 3. We can observe in Fig. 3a that the validation performance of CPAchecker is largely identical when comparing both formats. This is to be expected, as the only thing that CPAchecker extracts from the GraphML witnesses are the invariants and their locations, and this is also the information that is present in and extracted from the witnesses in format 2.0.

For UAutomizer, the new format 2.0 substantially improves both the speed of validation and the number of witnesses that can be validated. Besides aiding in verification of the original property during validation, a witness can also add additional obligations for the validator to validate. This is the case here, where the extensive automaton that is embedded into CPAchecker's witnesses in format 1.0 is harder to prove correct for UAutomizer than the much simpler set of invariants that is present in the witnesses in format 2.0. Table 4 shows numbers of confirmed and refuted witnesses.

Violation Witnesses. We present the results of our evaluation regarding RQ 1 for violation witnesses in Fig. 4, which is complemented by Table 5 with the concrete number of validated and refuted witnesses. For witnesses generated by CPAchecker (cf. Fig. 4a), Witch3 is able to confirm significantly more witnesses in the new format than Symbiotic-Witch2 is able to confirm in the old

Table 4: Correctness witnesses produced by CPAchecker: Validation with CPAchecker and UAutomizer

		Witnesses v1.0		Witnesses v2.0	
Validator	Witnesses	Confirmed	Refuted	Confirmed	Refuted
CPAchecker	6 729	4 685	0	4 741	0
UAutomizer	6 729	2 478	109	2 959	2



(a) Validation of violation witnesses generated by CPAchecker

(b) Validation of violation witnesses generated by Symbiotic

Fig. 4: Violation witnesses: Quantile plots for the time taken for validation of the old and the new witnesses generated by two different verifiers for two different validators

format. Due to the large number of features and underspecified semantics of the GraphML format, Symbiotic-Witch2 does not support all the attributes used in the GraphML witnesses. Ignoring these features leads to a larger state-space that needs to be explored during validation, which results in more timeouts, or misinterpreting information in the witness and missing the described error. This is not the case for Witch3 as it supports the full set of features of the new format and makes use of all the information provided in the witness. For CPAchecker there is still a relatively small performance gap between validation with the new and old format. This is not surprising, as the GraphML-based format is inspired by the specification-automata language that CPAchecker uses internally, so achieving a similar performance requires still some more engineering.

For witnesses generated by Symbiotic (cf. Fig. 4b), we can observe that the number of new witnesses confirmed by Witch3 is almost the same as the number of old witnesses confirmed by Symbiotic-Witch2. Thus, both validation approaches are very close when it comes to effectiveness. This is also the case for

Table 5: Results of validating CPAchecker's and Symbiotic's **violation** witnesses with different validators; Witch stands for Symbiotic-Witch2 when validating old witnesses, and for Witch3 when validating new witnesses.

			Witnesses v1.0		Witnesses v2.0	
Verifier	Validator	Produced	Confirmed	Refuted	Confirmed	Refuted
CPAchecker	CPAchecker	2 011	1 880	35	1 657	9
CPAchecker	Witch	2 011	798	0	1 312	17
Symbiotic	Witch	1 556	1 533	5	1 516	0
Symbiotic	CPAchecker	1 556	1 319	29	1 315	27

Table 6: Different attributes of correctness witnesses in version 1.0 and 2.0 generated by CPAchecker

	Witnesses v1.0			Witnesses v2.0		
Attribute	Min	Median	Max	Min	Median	Max
Length in Lines of Code	53	1 536	1 014 533	18	28	1 058
Size in kB	3	52	35 573	1	1	965
Number of Nodes	3	114	26 899	-	-	-
Number of Edges	2	198	142 016	-	-	-
Number of Invariants	0	1	162	0	1	104

CPAchecker, which manages to confirm almost the same number of witnesses in both formats.

The new format for correctness witnesses does not reduce validation performance, for UAutomizer it shows a significant advantage over the old format. For violation witnesses, Witch3 handling the new format performs better than Symbiotic-Witch2 on the old format, while there is still room for improvement of CPAchecker as it performs slightly better on the old format.

3.2 Evaluation results for RQ 2 (Witness Readability)

Another important question is concerned with the attributes corresponding to the readability of the files encoding the witnesses. In particular, we are interested in the size and length of the witnesses, since this has a large effect on how easy they are to be read and understood by humans and machines.

Table 6 provides an overview of different attributes of the two versions of witnesses produced by CPAchecker for correctness. Table 7 does the same for violation witnesses produced by CPAchecker and Symbiotic. Some attributes are only applicable to one of the two versions of witnesses.

For correctness witnesses we can see that the new witnesses are usually very small in comparison to the old witnesses. This is because the new witnesses encode only the invariants, while the old witnesses encode information about the control-flow of the program. One explanation for the difference is that witnesses in version 1.0 roughly scale with the size of the program. While witnesses in version 2.0 scale

Table 7: Different attributes of violation witnesses in version 1.0 and 2.0 generated by CPAchecker and Symbiotic

	Witnesses v1.0			Witnesses v2.0		
Attribute	Min	Median	Max	Min	Median	Max
Length in Lines of Code	12	372	258 730	27	171	114 460
Size in kB	2	14	9 098	1	6	3 071
Number of Nodes	1	38	28 304	-	-	-
Number of Edges	0	42	28 793	-	-	-
Number of Waypoints	-	-	-	1	13	9 537

only with respect to the amount of invariants, which for CPAchecker is roughly correlated to the amount of function calls and loops. As we saw in Sect. 3.1, this extra information is not necessarily relevant for validation.

For violation witnesses, we see that, apart from a small factor due to overhead in describing the automaton, both formats are similar in all metrics. This is not surprising, as both formats encode similar information about an error path. Therefore, they both roughly scale with the amount of assumption for nondeterministic variables and the amount of branching decisions in the error path.

The tables show that the new witnesses are usually much shorter than the old witnesses. As we have seen in Sect. 3.1, this does not have a negative impact on the validation performance, since the information most relevant for validation is retained. Having less information makes it much easier for a verification engineer to understand the witness and use it in some further processing steps.

In summary, witnesses in version 2.0 are generally much smaller and easier to read than witnesses in version 1.0, while retaining all important data.

3.3 Threats to Validity

Internal Validity. We used the benchmarking framework BenchExec [33] to run the experiments, which uses the most modern Linux features for reliable benchmarking. This tool also makes sure to never run two different executions on the same physical core, in order to avoid interference of shared computing resources. Our validation tools might contain bugs, which could lead to wrong conclusions, however, our claim is that the new format works already sufficiently well to serve as an alternative format.

External Validity. The conclusions about the validators might not hold for other validators that will be developed in the future, also, witnesses generated by other verifiers might have different characteristics. However, other tools are not expected to deviate much from the presented witnesses, because they would serve the same purpose of testifying the bug or proof. Our experiments were done on a

large benchmark set, which is also used in competitions, but it could still be the case that there are witnesses and programs for which the results presented are not applicable. Since extending and improving the witness format is an ongoing process, we expect that if this is the case, it will be adequately addressed in the future.

4 Version 1.0 vs. Version 2.0

The witness format 2.0 is closely tied to the actual program syntax. While the format 1.0 uses an automaton largely independent of the program syntax and closely tied to the program representation as control-flow automata internally used by some verifiers. Due to this, the format 2.0 is more succinct, has well-defined semantics, and is easier to understand by humans. On the other hand, format 1.0 is more expressive, for example it can define different loop invariants for the same loop, when two different paths are taken to reach the loop.

Currently, the format 2.0 has the same practical limitations as the format 1.0. In the case of correctness witnesses, they have not yet been defined for concurrency safety, memory safety and for termination. Violation witnesses have not yet been defined for concurrency safety. There are also features which are not yet supported by the new format but which are straightforward extensions, such as support for Java and violation termination witnesses. Extending witnesses to be able to validate more programs and specifications is ongoing work, we expect that the simplification of the syntax and clarification of the semantics with format version 2.0 will make it easier to extend the format in the future.

In order to validate our concept of the new format, we reported our initiative to the SV-COMP community, and the jury made a decision to immediately include the new format as an alternative to the existing format, in order to quickly adopt it and improve the state of the art. This was seen in SV-COMP 2024 [32], where 8 verifiers and 4 validators supported the correctness witnesses v2.0 and 2 verifiers and 2 validators supported the violation witnesses v2.0. Table 8 shows all these tools and their support of witnesses formats in detail.

This also shows the large interest the software verification community has in the new format, since the first mention of the format for correctness witnesses was only in September 2021[4] and the work on the violation witnesses part of the new format started only in April 2023.

5 Conclusion

Verification witnesses are an important part of the software-verification ecosystem. Just like verification tools, specification formats, and witness validators, there is also a need to improve the *format* for verification witnesses. This paper introduces the witness format version 2.0, which changes the container format from GraphML to YAML, has more concise data representation, and has a clearly

[4] https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/merge_requests/44

Table 8: Tools with some support of witnesses format 2.0 and their abilities to generate/verify correctness/violation witnesses in format 1.0/2.0 in SV-COMP 2024; tools where the support of witness format 2.0 was implemented by the authors of this paper are typeset in bold

	Witness Generation				Witness Validation			
	Correctness		Violation		Correctness		Violation	
Tool	v1.0	v2.0	v1.0	v2.0	v1.0	v2.0	v1.0	v2.0
CPAchecker	•	•	•	•	•	•	•	•
Symbiotic	•		•	•				
Symbiotic-Witch2							•	
Witch3								•
UAutomizer	•	•	•		•	•	•	
UKojak	•	•	•					
UTaipan	•	•	•					
UGemCutter	•	•	•					
Mopsa	•	•				•		
CPV	•	•	•					
Goblint	•	•				•		

defined semantics independent from control-flow automata. Besides describing the syntax and semantics of the new format, we also evaluated the effectiveness and efficiency induced by the new format. In sum, the new witnesses are much smaller and the experimental results show a significantly improved confirmation rate for some validators: using the new format, UAutomizer can confirm 481 more correctness witnesses (Table 4) and Witch3 can confirm 514 more violation witnesses (Table 5). Furthermore, shortly after we proposed this new format, already seven other tools support the format, which is an indicator that the developers value the new format.

Data-Availability Statement. A reproduction package (that includes all software and data that we used for our experiments) is available on Zenodo [34].

Funding Statement. P. Ayaziová and J. Strejček were supported by the Czech Science Foundation grant GA23-06506S. D. Beyer, M. Lingsch-Rosenfeld, and M. Spiessl were supported by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 496588242 (IdeFix).

References

1. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351

3. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
4. Beyer, D., Strejček, J.: Case study on verification-witness validators: Where we are and where we go. In: Proc. SAS. pp. 160–174. LNCS 13790, Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_8
5. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
6. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
7. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064
8. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Proc. CAV (2). pp. 502–509. LNCS 9780, Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_28
9. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
10. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM. p. 111–128. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
12. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing. pp. 501–512. LNCS 2265, Springer (2001). https://doi.org/10.1007/3-540-45848-4_59
13. 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
14. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (competition contribution). In: Proc. TACAS (2). pp. 462–467. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32
15. Ayaziová, P., Strejček, J.: Symbiotic-Witch 2: More efficient algorithm and witness refutation (competition contribution). In: Proc. TACAS (2). pp. 523–528. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
16. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009
17. Turing, A.: Checking a large routine. In: Report on a Conference on High Speed Automatic Calculating Machines. pp. 67–69. Cambridge Univ. Math. Lab. (1949), https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8
18. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_6
19. Sternagel, C., Thiemann, R.: The certification problem format. In: Proc. UITP. pp. 61–72. EPTCS 167, EPTCS (2014). https://doi.org/10.4204/EPTCS.167.8

20. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine **33**(1) (2012)
21. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **1610**(06229) (October 2016)
22. Wetzler, N., Heule, M.J.H., Jr., W.A.H.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: Proc. SAT. pp. 422–429. LNCS 8561, Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31
23. Bury, G., Bobot, F.: Verifying models with Dolmen. In: Proc. SMT Workshop. CEUR Workshop Proceedings, CEUR (2023)
24. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Proc. ASE. pp. 200–205. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115633
25. OASIS: Static analysis results interchange format (sarif) version 2.0 (2019)
26. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
27. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
28. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
29. Ayaziová, P., Strejček, J.: Witch 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
30. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate automizer and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
31. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2024). Zenodo (2024). https://doi.org/10.5281/zenodo.10669737
32. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
33. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
34. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Reproduction package for SPIN 2024 article 'Software verification witnesses 2.0'. Zenodo (2024). https://doi.org/10.5281/zenodo.10826204