Verification Witnesses

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

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

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

Additional Key Words and Phrases: Violation Witness, Correctness Witness, Witness Validation, Software Verification, Program Analysis, Model Checking, Data-Flow Analysis, Formal Methods, Certifying Algorithm

ACM Reference Format:
Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. 2022. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2022), 69 pages. https://doi.org/10.1145/3477579

This article builds on concepts and techniques that were introduced in previous articles: Proc. FSE 2015 [25], Proc. FSE 2016 [23], and Proc. TAP 2018 [26].

Authors' addresses: Dirk Beyer, LMU Munich, Oettingenstraße 67, Munich, 80538, Bayern, Germany; Matthias Dangl, LMU Munich, Oettingenstraße 67, Munich, 80538, Bayern, Germany; Daniel Dietsch, University of Freiburg, Georges-Köhler-Allee 52, Freiburg, 79110, Baden-Württemberg, Germany; Matthias Heizmann, University of Freiburg, Georges-Köhler-Allee 52, Freiburg, 79110, Baden-Württemberg, Germany; Thomas Lemberger, LMU Munich, Oettingenstraße 67, Munich, 80538, Bayern, Germany; Michael Tautschnig, Queen Mary University of London, Mile End Road, London, E1 4NS, United Kingdom.

1049-331X/2022/1-ART1
https://doi.org/10.1145/3477579

Contents

1 INTRODUCTION

The omnipresent dependency on software in society and industry makes it necessary to ensure reliable and correct functioning of the software. This trend will continue and become even more important in the future. During the last decade, various conceptual breakthroughs in verification research were achieved, and, as showcased by the annual TACAS International Competition on Software Verification (SV-COMP 2012–2020)[1] [9, 10, 11, 12, 13], many successful software verifiers are already available.

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

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

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

Violation witnesses [25] are verification witnesses that document bugs detected by a verifier and address the problem of false alarms that imprecise verification tools sometimes produce: Formerly, a verification tool reported found bugs as error traces in a tool-specific manner; those bug reports were often difficult to read and understand, and therefore hardly usable. As a consequence, determining whether the reported bug was a false alarm that could be ignored by the developer or described an actual programming error that needed to be fixed was a tedious manual process. Exchangeable

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

violation witnesses resolve this issue, because several validators have been developed to use these witnesses to validate verification results [25, 26, 44, 122] and the general syntax of the exchange format allows new tools for presentation to be developed and used [22, 114].

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

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

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

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

2 BACKGROUND

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

2.1 Program Representation using Control-Flow Automata

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

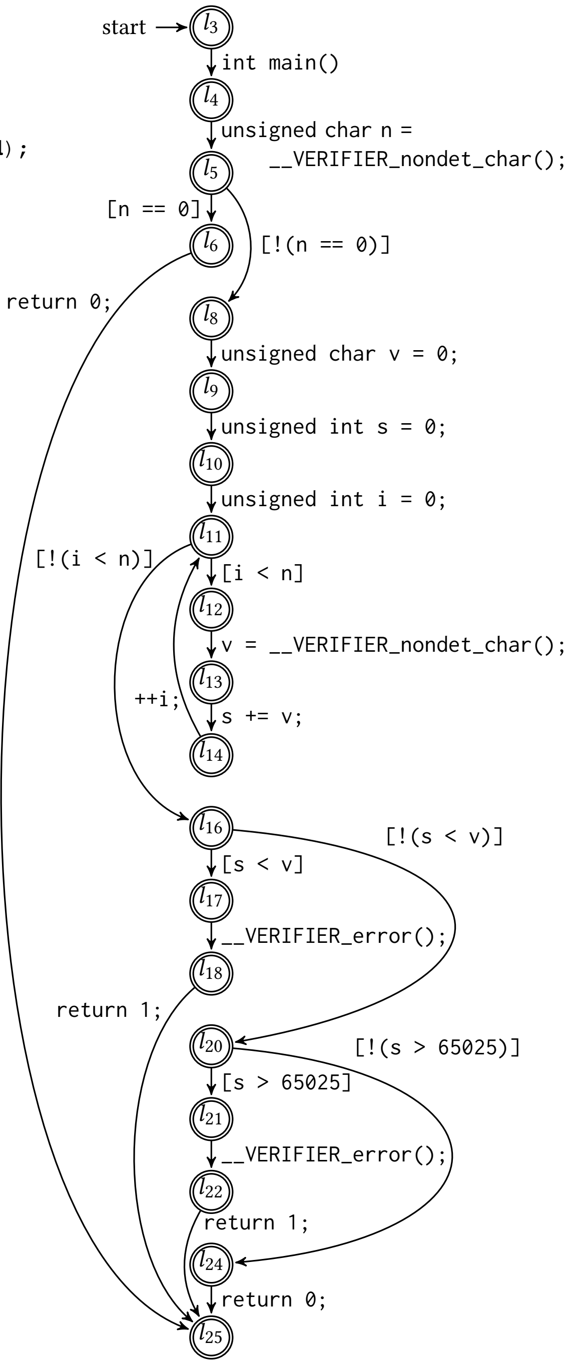
A sequence $\langle(l_0, op_1, l_1), \ldots\rangle$ of consecutive edges from G is called *program path* if it starts at the initial location, i.e., $l_0 = l_{init}$. A *test vector* [19] specifies the input values to a program. A program path is called *feasible*, if a test vector exists for which this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

Example 1 (Program and Control-Flow Automaton)**.** Figure 1 shows the source code (Fig. 1a) of an example C program that computes the sum of a number of input values and the corresponding CFA (Fig. 1b). Location $l_{init} = l_3$ is the initial location of this program. The program contains four variables: `n` is the number of values to sum up, `v` is used to hold input values for the computation, `s` is the aggregation variable for the sum, and `i` is a loop counter. The type of the variables `n` and `v` is `unsigned char`, the type of the variables `s` and `i` is `unsigned int`, and for all our examples we will assume a data-type model where the type `unsigned char` has a bit width of 8, such that the values of this type range between `0` and `255`, and the type `unsigned int` has a bit width of 32, such that the values of this type range between `0` and `4294967295`. The CFA starts with the function entry in line 3, modeled by the edge from l_3 to l_4. The next CFA edge, (l_4 to l_5), shows that the number of values to sum up, `n`, is initialized via the input function `__VERIFIER_nondet_char(void)` in line 4. In line 5, the program checks the value of `n` and immediately terminates in line 6 if it is `0`, which indicates that there are no values to be summed up. This check is modeled in the CFA by the branching at l_5, which goes from l_5 over l_6 to l_{25} in the early-return case and from l_5 to l_8 in the other case. In lines 8–10, the remaining variables `v`, `s`, and `i` are all initialized to `0`, which corresponds to the CFA edges from l_8 to l_{11}. The loop that computes the sum of input values read via `__VERIFIER_nondet_char(void)` in lines 11–15 is modeled by the CFA nodes l_{11}, l_{12}, l_{13}, and l_{14}, with l_{11} being the loop head. After the

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

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

(a) Program source code



(b) CFA

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

loop, there are two assertions: In line 16, the program checks that the sum is not less than the last value added to it, which seems like a sensible requirement given that all added values are non-negative. If the check fails, the program calls the function `__VERIFIER_error(void)` in line 17 to indicate an error and terminates in line 18. This check is modeled in the CFA by the branching at l_{16}, which goes from l_{16} over l_{17} and l_{18} to l_{25} in the failing case and from l_{16} to l_{20} in the other case. In line 20, the program checks that the sum is not greater than `65025`, which is the product of the maximum value of `n` and therefore the maximum number of values that are added up, `255`, and the maximum value of each value being added up, which is also `255`. If the check fails, the program calls the function `__VERIFIER_error(void)` in line 21 to indicate an error and terminates in line 22. This check is modeled in the CFA by the branching at l_{20}, which goes from l_{20} over l_{21} and l_{22} to l_{25} in the failing case and from l_{20} to l_{24} in the other case, where the program terminates with return code 0 (success).

2.2 Configurable Program Analysis

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

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

2.2.1 CPA Algorithm.

The CPA algorithm takes a CPA and an initial abstract state as input (Alg. 1). Essentially, the algorithm performs a classic fixed-point iteration by computing successor states of reached abstract states until the set waitlist of unprocessed states is empty (i.e., until all reachable abstract states have been completely processed) and returns the set reached of reachable abstract states. In each major iteration, the algorithm takes one abstract state e from the waitlist, computes all its abstract successors, and processes each of them separately: For each successor abstract state e' the algorithm uses the operator merge to check if an already explored abstract state e'' with which the successor abstract state e' should be merged exists in the set reached of reached states (e.g., at meet points where the control flow of different paths meets after completed branching). If the operator merge decides that the two abstract states should be combined, then the

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

```
Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \text{merge}, \text{stop})$,
    where $E$ is the set of elements of the semilattice of $D$,
    and an initial abstract state $e_{init} \in E$,
Output: a set of reachable abstract states
Variables: two sets reached and waitlist of elements of $E$
 1: reached := $\{e_{init}\}$
 2: waitlist := $\{e_{init}\}$
 3: while waitlist $\neq \emptyset$ do
 4:   pop $e$ from waitlist
 5:   for all $e'$ with $e \rightsquigarrow e'$ do
 6:     for all $e'' \in$ reached do
 7:       $e_{new}$ := merge($e', e''$)
 8:       if $e_{new} \neq e''$ then
 9:         waitlist := (waitlist $\cup \{e_{new}\}$) $\setminus \{e''\}$
10:         reached := (reached $\cup \{e_{new}\}$) $\setminus \{e''\}$
11:     if not stop($e'$, reached) then
12:       waitlist := waitlist $\cup \{e'\}$
13:       reached := reached $\cup \{e'\}$
14: return reached
```

existing abstract state e'' is substituted by the new, merged abstract state e_{new} in both sets reached and waitlist. The stop operator implements the detection of a fixed point. The CPA algorithm uses it to check if the new abstract state e' is already covered by an existing abstract state in the set reached, and only if the result is negative it inserts the new abstract state e' into the work sets waitlist and reached, i.e., only if this is necessary to explore this abstract state further.

2.2.2 Composite CPA. A *Composite CPA* [34] can be used to combine a set of CPAs into a single, composite, CPA. An abstract state of the Composite CPA is a tuple composed of one component abstract state for each component CPA and the operators merge and stop are defined to delegate to the component CPAs' respective operators, such that the merge operator combines abstract states according to how the components' merge operators combine the component abstract states, and the operator stop only returns *true* if all components agree that their component abstract states are already covered by their respective existing component abstract states in the set reached.

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

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

3 CONCEPTS

The purpose of verification witnesses is to represent information about verification results in such a manner that it is machine-readable, reproducible, and exchangeable between verification tools. There are two types of verification witnesses: Violation witnesses, which represent error paths that violate a specification, and correctness witnesses, which represent the artifacts of a proof that a program satisfies a specification. In this section, we present the basic concepts for verification witnesses whereas the specifics of violation witnesses and correctness witnesses will be discussed in a later section on design and implementation (Sect. 4).

3.1 Protocol Automata

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

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

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

We define a number of properties for protocol automata:

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

Stutter-Enabled State. A state $q \in Q$ is called *stutter-enabled*, if there is a special self-transition $q \xrightarrow{o/w} q \in \delta$ (with the special guard symbol "*o/w*" short for "otherwise"), which is defined as follows: Let $\delta_{q,other}$ be the set of all outgoing transitions of q except those with the guard *o/w*, i.e., $\delta_{q,other} = \left\{ q \xrightarrow{\sigma} q' \;\middle|\; \sigma \neq o/w \right\}$. The transition $q \xrightarrow{o/w} q$ is a self-transition where the state-space guard ψ is *true* and the source-code guard S matches the set of all CFA edges that are either (a) not matched by the source-code guard of any other outgoing transition of q or (b) are matched by the source-code guard of some other outgoing transition of q that also matches a successor CFA edge. Thus, a stutter-enabled state ensures that for every CFA edge, there is always at least one outgoing transition of the state where the source-code guard matches the transition, which is

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

a requirement of our witness automata (see Sect. 3.1.2). Moreover, the mechanism can be used to support nondeterminism, because if there is a transition with a source-code guard that ambiguously matches two (or, transitively, more) consecutive CFA edges, there is also a matching self-transition that does not impose a state-space guard. More formally, the transition $q \xrightarrow{o/w} q$ is equivalent to a transition $q \xrightarrow{(S_{q,stutter}, true)} q$ with $S_{q,stutter} = (G \setminus S_{q,other}) \cup S_{q,ambig}$, where $S_{q,other}$ is the set of all CFA edges that are matched by the source-code guard S' of any other outgoing transition of q, i.e., $S_{q,other} = \bigcup \left\{ S' \;\middle|\; q \xrightarrow{(S',\cdot)} q' \in \delta_{q,other} \right\}$, and $S_{q,ambig}$ is the set of those CFA edges matched by any other outgoing transition of q that have a successor CFA edge that is matched by the same transition, i.e., $S_{q,ambig} = \left\{ s \in G \;\middle|\; \exists q \xrightarrow{(S,\cdot)} q' \in \delta_{q,other},\ s, s' \in S : s = (\cdot, \cdot, l_b) \wedge\ s' = (l_b, \cdot, \cdot) \right\}$.

3.1.1 Control-Flow Automata.

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

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

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

3.1.2 Witness Automata.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

with $\sigma_i = (S, \psi)$ and $(l_{i-1}, op_i, l_i) \in S$ and all variable assignments that are attached to the i-th program location of $\widehat{p}$ satisfy ψ.

3.1.7 Acceptance. Protocol automata provide flexibility regarding the acceptance criterion in that they allow a choice: If the goal is to accept *finite* runs (e.g., for the witness-based validation of verification results for reachability problems, as in the examples in this article), a protocol automaton A can be defined to accept the run $\langle \ldots, q_{n-1} \xrightarrow{\sigma_n} q_n \rangle$ if $q_n \in F$. If the goal is to accept *infinite* runs (e.g., for the witness-based validation of verification results for termination problems), we can define the protocol automaton A as a Büchi automaton that accepts an infinite run ρ if there exists a control state $q \in F$ that occurs infinitely often in ρ. We say A accepts a program path p if there exists an accepting run of A for p. The projection of an accepted finite run to the sequence $\langle \sigma_1, \ldots, \sigma_n \rangle$ of its alphabet symbols is called *accepted word*, as is the projection of an accepted infinite run to the sequence $\langle \sigma_1, \ldots \rangle$ of its alphabet symbols. The set of all accepted words of A defines the *language* $\mathcal{L}(A)$.

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

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

3.2 Automata Representations

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

Table 1. Mapping artifacts related to software analysis to their representing types of automata and to examples of analyses that use these artifacts

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

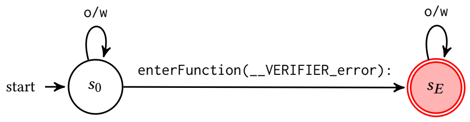


Fig. 2. Specification that forbids calls to the function `__VERIFIER_error(void)`, represented as observer automaton that accepts all program paths that enter control state s_E, i.e., violate the specification

3.2.1 *Specifications Represented by Observer Automata.*

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

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

3.2.2 *Violation Witnesses Represented by Witness Automata.*

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

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

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

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

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

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

3.3 Automaton CPA: A Configurable Program Analysis for Protocol Automata

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

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

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

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

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

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

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

3.4 Constructing Witness Automata from Proofs

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

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

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

3.5 Application Scenarios

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

3.5.1 Verification with Witnesses. Good practice requires a verifier, whenever it reaches a conclusion regarding a given verification task, to produce a verification witness that provides information about the verification result. The purpose of the verification witness is to document the verification result and to make valuable verification artifacts available for reuse instead of leaving unused the effort the verifier has already spent on them. The primary use case we discuss in this article is witness-based result validation. Another use case is the visualization of verification results [22, 114]. Figure 3 illustrates the process of verification with witnesses, and it also shows one key feature of the concept of having a common representation of verification results: there is no risk of technology lock-in, because the verifiers are interchangeable according to the needs of the user.

3.5.2 Witness-Based Result Validation. A witness-based result validator can independently re-establish a verification result of a verifier using the guidance of a verification witness. We describe a program analysis for this purpose using the CPA concept by configuring a Composite CPA with the following components: One component is an Automaton CPA that performs a protocol analysis (cf. Sect. 3.3) for a witness automaton (which we also call witness analysis, because it simulates the witness automaton). One of the other component CPAs is an Automaton CPA that performs an observer analysis (cf. Sect. 3.3) and encodes the specification, which is represented by an observer automaton (cf. Sect. 3.1.3), i.e., which only observes but does not restrict how the program's state space is explored by the program analysis. The composed program

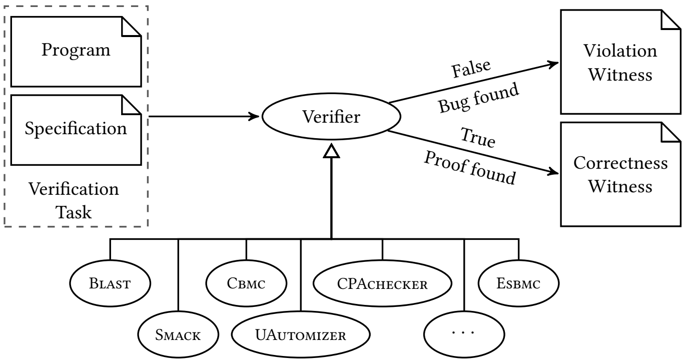


Fig. 3. Software verifiers produce witnesses

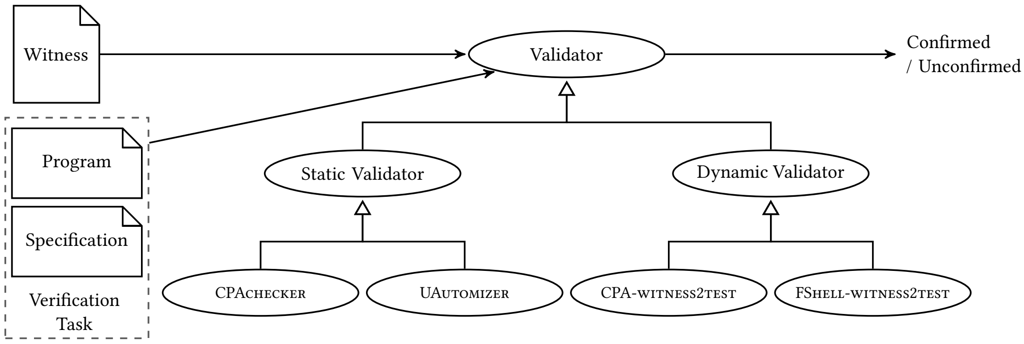


Fig. 4. Witness-based result validation

analysis only considers specification violations signaled by this CPA if the CPA that simulates the witness automaton agrees, i.e., if both the specification automaton *and* the witness automaton accept the corresponding run. Another component CPA is the Location CPA, which is used to track the program counter, i.e., the location in the CFA, for the analysis. Further components can be added to the composition, for example to track information about the values of program variables. These components can then use the operator $\downarrow$ of the Composite CPA to compute the intersection of their component abstract states with the state-space guards from the witness automaton to achieve a restricted, more precise state space, or they can check the validity of the state invariants of the witness automaton, and, if successful, use these invariants as proof lemmas. Figure 4 illustrates the process of witness-based result validation and shows four existing implementations of validators. A verification result is confirmed by a witness-based result validator if the validator is able to re-establish the verification result.

We illustrate witness-based result validation with Fig. 5: First, the verifier receives the verification task $P \models \varphi$ as input and constructs a proof π for proving or disproving the statement. In the former case the verifier produces a correctness witness (which contains invariants) and in the latter case the verifier produces a violation witness (which contains an error path). Second, a validator receives the same verification task $P \models \varphi$ and the witness as input but constructs a new proof π' which might be different from the verifier's proof π. However, the validator is allowed to look into the verification witness in order to obtain information to support the proving process. If it receives a correctness witness as input, then it tries to use the invariants (i.e., using

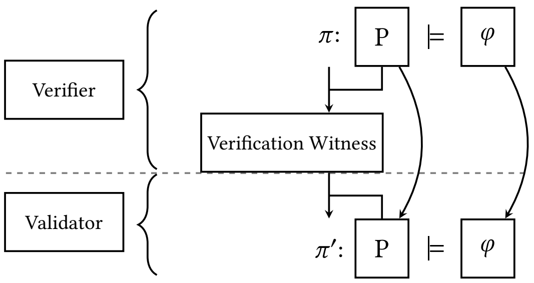


Fig. 5. In witness-based result validation, a verification witness is produced by a verifier that is able to find a proof π that proves or disproves $P \models \varphi$. The verification witness carries information that may guide the validator to find its own proof π' that also proves or disproves $P \models \varphi$.

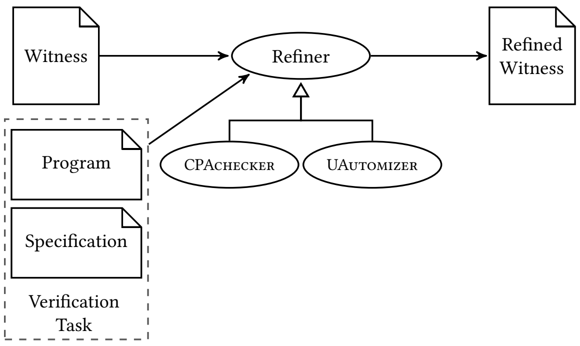


Fig. 6. Witness refinement

inductiveness checks); if it receives a violation witness as input, then it tries to use the error path (i.e., using a simulation to check feasibility).

3.5.3 Witness Refinement. *Witness refinement* is the iterative process of improving witnesses, by augmenting the verification artifacts represented by an input witness using new, potentially more detailed, information computed by a witness-based result validator. This process combines the concept of witness-producing verification with the concept of witness-based result validation. Figure 6 illustrates the process of witness refinement and shows the two existing implementations of witness refinement. We call a tool that is able to perform witness refinement a witness refiner. By using the common exchange format for verification witness, witness refinement can be applied using any desired sequence of witness refiners that are available to a user.

3.5.4 Witnesses for Concurrent Systems. For the ease of presentation, we restrict our descriptions to non-concurrent systems. The presented concepts, however, are also applicable to concurrent systems [29]: State-space guards and state invariants can be specified for specific threads. In a context-bounded view of a concurrent system (which is sufficient to describe a violation of a reachability property, for example), we can consider a *concurrent CFA* as a product of the original CFA and the (maximum) number of concurrent threads, and describe a schedule by using source-code guards to specify the set of CFA edges in the concurrent CFA as a combination of a set of CFA edges from the original CFA and the set of potentially active threads.

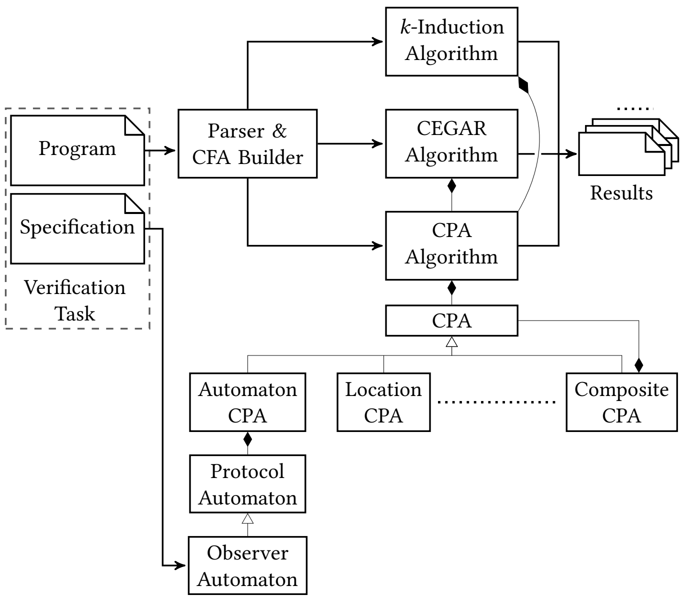


Fig. 7. Architecture of CPAchecker

3.5.5 Witnesses for Termination. While we use a reachability specification for our running example, the presented concepts are also applicable to termination. Instead of using the acceptance criterion for finite runs, we can treat the witness automata as Büchi automata and use the acceptance criterion for infinite runs, as described in Sect. 3.1.7.

4 DESIGN AND IMPLEMENTATION

In the following, we will explain how we use protocol automata to represent verification results and how we can validate and refine these verification results by applying witness-based verification-result validators.

4.1 Verifiers

Over the last decades, a multitude of automated software verifiers has been developed. We now give a brief introduction to two verification tools that we will use to generate witnesses for our evaluation and that three of the four violation-witness-based result validators and both of the correctness-witness-based result validators we present in the following sections are based on.

4.1.1 CPAchecker. The configurable software-verification framework CPAchecker [38] is based on configurable program analysis [30, 34] and supports many different verification approaches, such as predicate abstraction [21, 39, 77], lazy abstraction with interpolants [46, 109], k-induction [27, 69], bounded model checking [49], explicit-value analysis [42], and symbolic execution [40]. CPAchecker won the category *Overall* of the competition on software verification (SV-COMP) seven times from 2012–2021.[5]

Architecture. Its architecture is designed to explicitly reflect the concepts of configurable program analysis (cf. Sect. 2.2) in its components, as visualized by Fig. 7: The left side of the figure shows the input, which is a verification task that consists of a program and a specification. The source code of a verification task is parsed and converted into a CFA (cf. Sect. 2.1), the specification is

[5]https://cpachecker.sosy-lab.org/achieve.php

parsed and converted into an observer automaton (cf. Sect. 3.2.1). Then, the desired algorithm is run on the verification task. After the algorithm completes, the computed results, for example the verification outcome, are delivered. The core algorithm of the CPAchecker framework is the CPA algorithm (data-flow analysis/abstract interpretation [65, 101, 113]). As examples of CPAs implemented in CPAchecker, the figure shows the Location CPA and the Composite CPA, which were already introduced in Sect. 2.2, as well as the Automaton CPA, which is described in Sect. 3.3. The dotted line symbolizes that more CPAs that are not discussed in this article are available, for example, a CPA for predicate analysis [28] or a CPA for explicit-state model checking [42]. Besides the CPA algorithm, Fig. 7 depicts two additional algorithms as examples of further implemented verification approaches: counterexample-guided abstraction refinement (CEGAR) [59] and k-induction [27]. Both of these algorithms delegate parts of their work to the CPA algorithm. For example, CPAchecker can be configured to perform predicate abstraction [77] with counterexample-guided abstraction refinement by combining the CEGAR algorithm with a CPA that implements predicate abstraction. CPAchecker can also be configured to perform k-induction with auxiliary-invariant generation, by combining the k-induction algorithm with a program analysis that produces invariants. For a detailed and formal discussion on how these approaches are implemented in CPAchecker, we refer the reader to the literature [28] and briefly introduce the concept of k-induction, only because it is a non-trivial but integral component of one of the validation approaches that we will later present.

***k*-Induction.** To obtain unbounded proofs of safety, k-induction combines techniques from bounded model checking [50] with induction. Consider a verification task that contains an unbounded loop and a candidate invariant P for that task. It is possible (a) to check using a bounded model check with bound $k = 1$ whether a program path of length $k = 1$ exists for which P is violated, but this check cannot prove the absence of longer counterexample paths. However, it is possible (b) to prove that P is an invariant using induction if P is inductive, i.e., if P holds before a given loop iteration, P also holds after that iteration, by taking (a) as the base case of the induction proof and (b) as the inductive-step case.

An extension to greater values of k lifts (1-)induction to k-induction, where the invariant P is asserted not only before one loop iteration, but before each of k consecutive loop iterations in the step case to conclude that it also holds after the k-th loop iteration. For $k > 1$, $(k-1)$-inductiveness implies k-inductiveness. In practice, k-induction may therefore succeed more often to prove correctness than $(k-1)$-induction [127], because k-induction uses a stronger induction hypothesis. A drawback of k-induction is that the approach cannot succeed if P is not k-inductive for any k. It is therefore desirable to strengthen P with auxiliary invariants to try making the assertion inductive.

KI↫DF and KI↫KI [27] are two k-induction techniques that are implemented in CPAchecker and use auxiliary invariants. In both techniques, an invariant generator runs in parallel to the k-induction procedure and successively provides invariants that are then used to strengthen the induction hypothesis. As time progresses, stronger invariants are generated, until the auxiliary invariants sufficiently strengthen the induction hypothesis to successfully prove the invariant P by induction. In KI↫DF, the auxiliary-invariant generator is based on a data-flow analysis. Over time, the precision used by the analysis is increased, causing stronger invariants to be generated. In the k-induction technique KI↫KI, the auxiliary-invariant generator is itself based on k-induction and attempts to prove invariants from a set of candidate invariants (either derived from a template or provided by a user). As time progresses, more confirmed candidates may become available as auxiliary invariants, until the induction hypothesis is strong enough to prove the safety property.

Example 5 (Verification with CPAchecker). Consider a verification task consisting of the program shown in Fig. 1 and the specification from Fig. 2. No feasible path to the call of the function `__VERIFIER_error(void)` in line 17 exists, because after the loop, the sum `s` of non-negative summands is always at least as great as its last summand `v`. Since the type of `s` is `unsigned int`, no overflow is caused by computing and storing the sum of at most `255` (maximum value of `n`) values, each of which is at most `255` (maximum value of `v`) itself, which in total is at most `65025`. Consequently, no feasible path to the call of the function `__VERIFIER_error(void)` in line 21 exists either. Therefore, the program satisfies the specification. CPAchecker is able to prove this by applying the KI⟲KI technique for k-induction and using a template for linear inequalities to produce candidate invariants for KI⟲KI: Knowing that $0 \leq i \leq 255$ due to the types of `i` and `n`, CPAchecker can prove that the linear inequality $s \leq i \cdot 255$ is an invariant, and consequently also that $s \leq 65025$ (and therefore also that `s` cannot overflow), which proves the program safe.

4.1.2 *UAutomizer.*

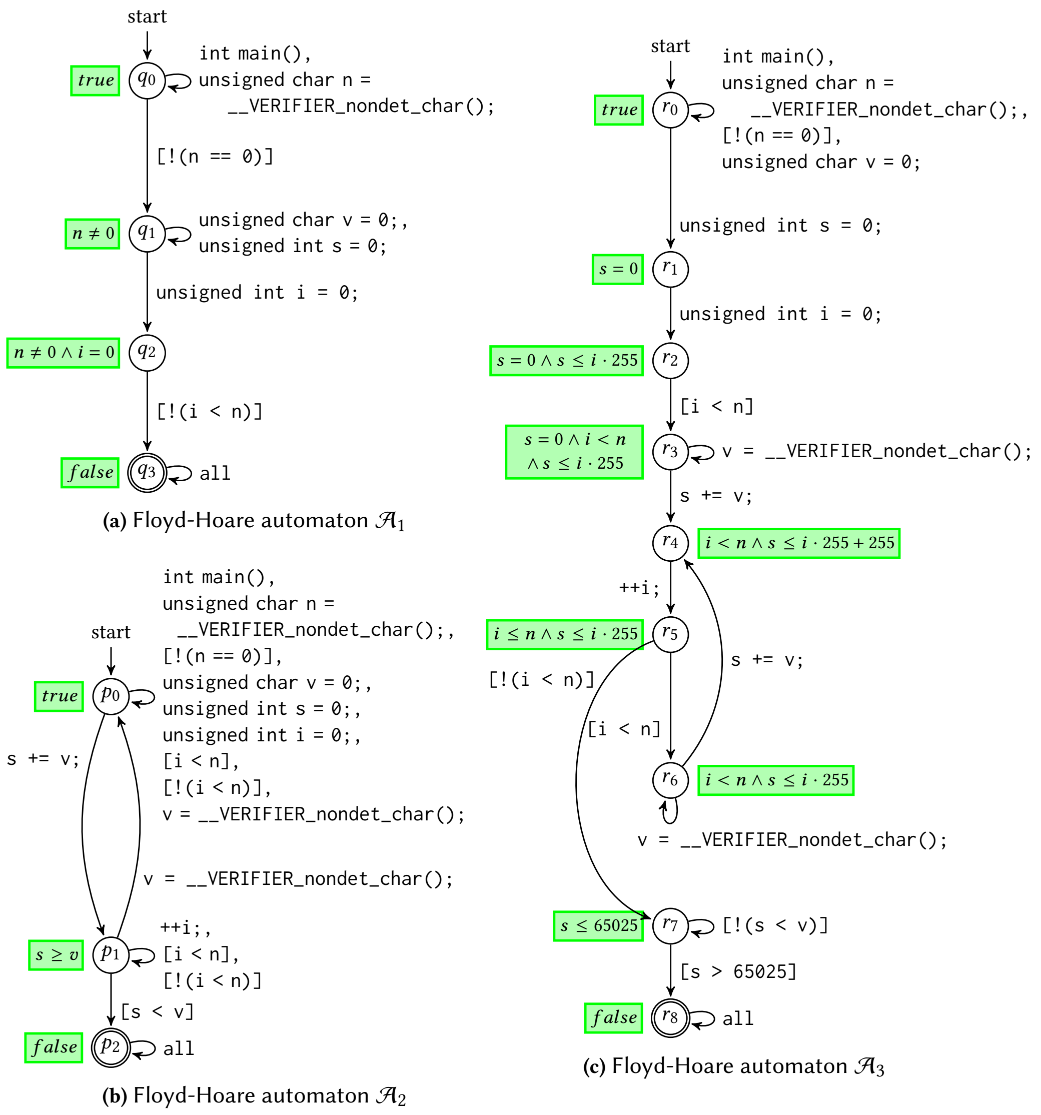
The automata-based verification approach of UAutomizer [84] constructs a correctness proof as a sequence of automata. UAutomizer uses the concept of *Floyd-Hoare automata* (cf. Sect. 3.1.5) instead of an ARG. UAutomizer won the category *Overall* of the competition on software verification (SV-COMP) two times from 2012–2020.[6]

Architecture. Like CPAchecker, UAutomizer transforms the given program into a CFA and the given specification into an observer automaton, and uses the specification to determine whether a program path is an error path. The automaton product of the CFA and the observer automaton for the specification yields a new CFA that describes a formal language over the alphabet G, where G is the set of control-flow edges, and where the accepting states are defined by the observer automaton for the specification. Hence, the words accepted by this automaton are exactly those paths through the program that violate the specification, i.e., the error paths (cf. Sect. 2.1).

Verification Using Floyd-Hoare Automata. To solve a verification task, UAutomizer iteratively constructs Floyd-Hoare automata instead of an ARG. The Floyd-Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are constructed such that each automaton accepts only words that correspond to infeasible paths. If, at some point of this iterative process, the union of the languages of these automata becomes a superset of the language accepted by the product of the CFA and the observer automaton for the specification, the verification is complete and the constructed Floyd-Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ represent a correctness proof for the program.

Given a CFA $\mathcal{A}_P$ and the Floyd-Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ from the above-mentioned correctness proof, the following approach can be used to construct invariants: First, an automata-theoretical product of the automata $\mathcal{A}_P$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is constructed. We make sure that in the product construction no Floyd-Hoare automaton is blocking and make each automaton total beforehand (that is, they are observer automata). The totalization is implemented by implicitly adding for each missing outgoing transition (in the automata in Fig. 8) an outgoing transition whose target is the initial state. Because the initial state is labeled by *true*, the totalized automata are still Floyd-Hoare automata. The states of the product are tuples of the form $(l, s_1, \ldots, s_n)$ where the first component is a program location of the CFA, and the $i + 1$-th component s_i is a state of the Floyd-Hoare automaton $\mathcal{A}_i$. Each tuple in the product is annotated by a formula that is the n-ary conjunction of the invariants of all s_i, that is, the annotation of the tuple $(l, s_1, \ldots, s_n)$ is the conjunction $\bigwedge_{i=1}^{n} \varphi_{s_i}$. Then, the invariant for a location l is computed as the disjunction of all annotations of those tuples that are reachable in the product and where the first component is location l.

[6]https://ultimate.informatik.uni-freiburg.de/Automizer


start
true
q_0
int main(), unsigned char n = __VERIFIER_nondet_char();
[!(n == 0)]
$n \neq 0$
q_1
unsigned char v = 0;, unsigned int s = 0;
unsigned int i = 0;
$n \neq 0 \wedge i = 0$
q_2
[!(i < n)]
false
q_3
all
start
true
p_0
int main(), unsigned char n = __VERIFIER_nondet_char();, [!(n == 0)], unsigned char v = 0;, unsigned int s = 0;, unsigned int i = 0;, [i < n], [!(i < n)], v = __VERIFIER_nondet_char();
s += v;
v = __VERIFIER_nondet_char();
$s \geq v$
p_1
++i;, [i < n], [!(i < n)]
[s < v]
false
p_2
all
start
true
r_0
int main(), unsigned char n = __VERIFIER_nondet_char();, [!(n == 0)], unsigned char v = 0;
unsigned int s = 0;
$s = 0$
r_1
unsigned int i = 0;
$s = 0 \wedge s \leq i \cdot 255$
r_2
[i < n]
$s = 0 \wedge i < n \wedge s \leq i \cdot 255$
r_3
v = __VERIFIER_nondet_char();
s += v;
r_4
$i < n \wedge s \leq i \cdot 255 + 255$
++i;
$i \leq n \wedge s \leq i \cdot 255$
r_5
[!(i < n)]
s += v;
[i < n]
r_6
$i < n \wedge s \leq i \cdot 255$
v = __VERIFIER_nondet_char();
$s \leq 65025$
r_7
[!(s < v)]
[s > 65025]
false
r_8
all

(a) Floyd-Hoare automaton $\mathcal{A}_1$

(b) Floyd-Hoare automaton $\mathcal{A}_2$

(c) Floyd-Hoare automaton $\mathcal{A}_3$

Fig. 8. Proof that the program whose CFA is depicted in Fig. 1 satisfies the specification depicted in Fig. 2. We construct a product of these automata in order to obtain invariants for the program.

Example 6 (Verification with UAutomizer). The three Floyd-Hoare automata depicted in Fig. 8 are a proof that the program whose CFA is depicted in Fig. 1 satisfies the specification depicted in Fig. 2. The observer automaton (Fig. 2) for the specification considers the locations after the function `__VERIFIER_error(void)` was called, i.e., l_{18} and l_{22}, accepting. The three Floyd-Hoare automata are a proof of correctness, because each word that is accepted by the CFA is also accepted by $\mathcal{A}_1$, $\mathcal{A}_2$, or $\mathcal{A}_2$.

Table 2. Reachable states in the product automaton of $\mathcal{A}_P$, $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ together with their annotation

State	Annotation
(l_3, q_0, p_0, r_0)	$true$
(l_4, q_0, p_0, r_0)	$true$
(l_5, q_0, p_0, r_0)	$true$
(l_8, q_1, p_0, r_0)	$n \neq 0$
(l_9, q_1, p_0, r_0)	$n \neq 0$
(l_{10}, q_1, p_0, r_1)	$n \neq 0 \land s = 0$
(l_{11}, q_2, p_0, r_2)	$n \neq 0 \land i = 0 \land s = 0 \land s \leq i \cdot 255$
(l_{16}, q_3, p_0, r_0)	$false$
(l_{12}, q_0, p_0, r_3)	$s = 0 \land i < n \land s \leq i \cdot 255$
(l_{13}, q_0, p_0, r_3)	$s = 0 \land i < n \land s \leq i \cdot 255$
(l_{14}, q_0, p_1, r_4)	$s \geq v \land i < n \land s \leq i \cdot 255 + 255$
(l_{11}, q_0, p_1, r_5)	$s \geq v \land i \leq n \land s \leq i \cdot 255$
(l_{12}, q_0, p_1, r_6)	$s \geq v \land i < n \land s \leq i \cdot 255$
(l_{13}, q_0, p_0, r_6)	$i < n \land s \leq i \cdot 255$
(l_{14}, q_0, p_1, r_4)	$s \geq v \land i < n \land s \leq i \cdot 255 + 255$
(l_{16}, q_0, p_1, r_7)	$s \geq v \land s \leq 65025$
(l_{17}, q_0, p_2, r_0)	$false$
(l_{20}, q_0, p_0, r_7)	$s \leq 65025$
(l_{21}, q_0, p_0, r_8)	$false$

Table 3. Invariants for the program depicted in Fig. 1

Location	Invariant
l_3	$true$
l_4	$true$
l_5	$true$
l_8	$n \neq 0$
l_9	$n \neq 0$
l_{10}	$n \neq 0 \land s = 0$
l_{11}	$(n \neq 0 \land i = 0 \land s = 0 \lor s \geq v \land i \leq n) \land s \leq i \cdot 255$
l_{12}	$(s = 0 \lor s \geq v) \land i < n \land s \leq i \cdot 255$
l_{13}	$i < n \land s \leq i \cdot 255$
l_{14}	$s \geq v \land i < n \land s \leq i \cdot 255 + 255$
l_{16}	$s \geq v \land s \leq 65025$
l_{17}	$false$
l_{20}	$s \leq 65025$
l_{21}	$false$

Intuitively, the Floyd-Hoare automaton $\mathcal{A}_1$ says that we cannot leave the while loop without passing the body at least once. The Floyd-Hoare automaton $\mathcal{A}_2$ says that after running the while loop (at least) once, the value of `s` is not smaller than the value of `v` and hence the program cannot reach the first call of the `__VERIFIER_error(void)` function. The Floyd-Hoare automaton $\mathcal{A}_3$ says that $s \leq i \cdot 255$ is a loop invariant and since the loop counter `i` is bounded by an unsigned char, the value of `s` is bounded by 65025 and the program cannot reach the second call of the `__VERIFIER_error(void)` function.

Table 2 shows the annotations for the reachable states in the product of the automata from Fig. 8, and Table 3 shows the invariants that we obtain for the program depicted in Fig. 1. We do not show the locations l_6, l_{24}, and l_{25}, because UAutomizer detects beforehand that these locations do not occur on any path from the initial location to an error location and assigns the invariant *true* to these locations. In Table 2 we leave out all successors of all tuples that are annotated with *false* (l_{18} and l_{22}). Because each Floyd-Hoare automaton has a self-loop for locations with the invariant *false*, each successor of a tuple annotated by *false* is also annotated by *false*.

4.2 Result Validation Based on Violation Witnesses

Violation witnesses are verification witnesses that represent error paths, i.e., paths through the program source code that violate the specification (Sect. 3.2.2).

4.2.1 Principles. A violation-witness based result validator can attempt to validate the verification result FALSE if the result is supported by a violation witness. In 2019, four implementations of such validators existed: CPAchecker, UAutomizer, CPA-witness2test, and FShell-witness2test. Conceptually, all of these validators are based on the principle of witness-based result validation as described in Sect. 3.5.2. We classify the four validators into two categories, namely the category of *static* (*model-checking-based*) validators, and the category of *dynamic* (*execution-based*) validators, as indicated in Fig. 4.

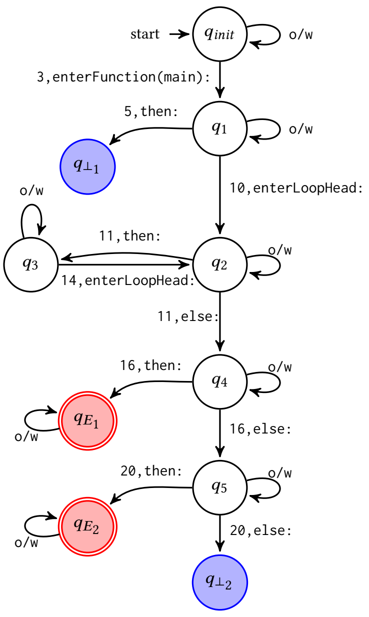
Static Validation. CPAchecker and UAutomizer are *static* validators, because they are based on purely static analysis and do not actually execute the program to confirm a violation. The advantages of static validators are that (1) they do not strictly require precise witnesses but can also be used with imprecise witnesses and can even be used to refine them, using witness refinement (cf. Sect. 3.5.3), because they can use model-checking techniques to detect invariants or even find concrete value assignments for program variables, (2) they can be used to validate results for verification tasks of systems with arbitrary target architectures, independent from the environment the validator is executed in, because they do not need to execute the analyzed program, and (3) they can be used for arbitrary specifications and are not limited to specifications with finite counterexamples (cf. dynamic validators). The main disadvantage of static validators is that accurately modeling all features of a complex programming language (such as C) is often difficult, and static validators therefore may exhibit the same imprecisions that also contribute to false alarms in verifiers. Thus, they may be less trustworthy than dynamic validators, which actually run the analyzed program to confirm a violation.

Example 7 (Violation-Witness Construction, Validation, and Refinement)**.** We illustrate how violation witnesses are constructed, validated, and refined across verifiers: We start with an overview of an example scenario and then describe the process of producing, consuming, and refining violation witnesses in more detail.

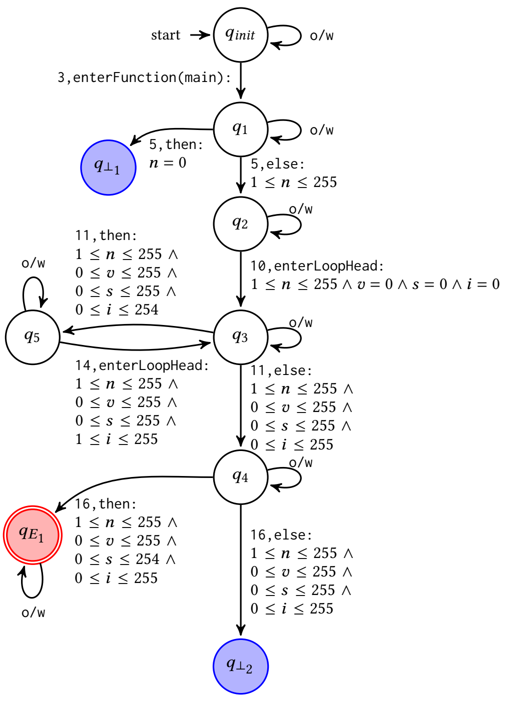
In this example, we first run three verifier instances in sequence. Each of them takes the verification task that consists of the program depicted in Fig. 9a and the specification shown in Fig. 2 as input, and produces a violation witness. The program in Fig. 9a differs from the program in Fig. 1a (and described in Sect. 2.1) in only one line: In line 9 of the original program, variable `s` is declared as type `unsigned int`, whereas in line 9 of the modified program, `s` is declared as type `unsigned char`; therefore, the only difference between the CFA of the original

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

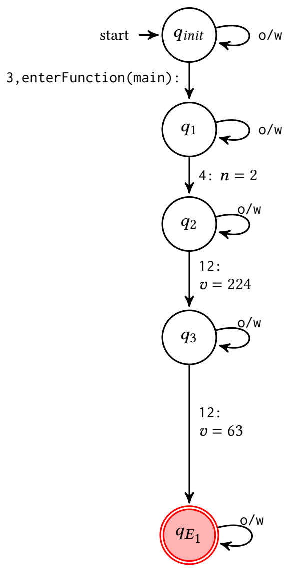
(a) Unsafe program linear-inequality-inv-b.c (adapted from Fig. 1a)



(b) Witness 1


start
q_{init}
o/w
3,enterFunction(main):
q_1
5,then: $n = 0$
$q_{\perp_1}$
5,else: $1 \le n \le 255$
q_2
10,enterLoopHead: $1 \le n \le 255 \wedge v = 0 \wedge s = 0 \wedge i = 0$
11,then: $1 \le n \le 255 \wedge 0 \le v \le 255 \wedge 0 \le s \le 255 \wedge 0 \le i \le 254$
q_5
q_3
14,enterLoopHead: $1 \le n \le 255 \wedge 0 \le v \le 255 \wedge 0 \le s \le 255 \wedge 1 \le i \le 255$
11,else: $1 \le n \le 255 \wedge 0 \le v \le 255 \wedge 0 \le s \le 255 \wedge 0 \le i \le 255$
q_4
16,then: $1 \le n \le 255 \wedge 0 \le v \le 255 \wedge 0 \le s \le 254 \wedge 0 \le i \le 255$
q_{E_1}
16,else: $1 \le n \le 255 \wedge 0 \le v \le 255 \wedge 0 \le s \le 255 \wedge 0 \le i \le 255$
$q_{\perp_2}$

(c) Witness 2



(d) Witness 3

Fig. 9. Example C program with a bug (a) and violation witnesses for it (b, c, and d)

program (cf. Fig. 1b) and the CFA of the modified program is that the label of the CFA edge between l_9 and l_{10} is changed to `unsigned char s = 0`. The specification requires that no call to the function `__VERIFIER_error()` must be reachable from the program entry. The program violates this specification: Recall from Sect. 2.1 that the program attempts to compute the sum of a number of input values. However, the variable `s` that is used to store the computed sum is now declared to be of type `unsigned char`, which in our setting means that it is only 8 bits wide and can only store values between `0` and `255`. It is therefore not suitable for the task of storing the sum of up to `255` values in the range of `0` to `255` and is susceptible to arithmetic overflows. As a result, it is possible —depending on the actual input values— that in line 16, the condition `s < v` actually holds and the function `__VERIFIER_error()` is called, thereby violating the specification.

In this example scenario, all three verifiers are configured as a composite CPA (cf. Sect. 2.2.2). The first verifier runs an analysis that considers only control-flow information and does not track any variable values, and produced Witness 1 (Fig. 9b). The second verifier then takes this witness and the verification task as input, runs an analysis based on an interval domain [64], and produces the violation witness in Fig. 9c. In the third step, we run a test-case generator [19, 26, 35] that takes the violation witness from Fig. 9c and the verification task as input and produces the test vector that is represented by the witness in Fig. 9d.

Verification. The Composite CPA used by the first verifier is composed of a Location CPA that tracks the program counter and an observer analysis (i.e., an Automaton CPA for an observer automaton, cf. Fig. 7) that tracks the control state of the observer automaton for the specification. Abstract states of this composite analysis are tuples $(l, (s, \psi_s))$, where l represents the current location in the CFA, i.e., the component abstract state tracked by the Location CPA, and (s, ψ_s) is the component abstract state tracked by the observer analysis, which consists of the current control state s of the observer automaton for the specification and the current state-space restricting condition ψ_s. Because the specification is represented as an observer automaton, the observer analysis does not restrict the state space. This analysis will detect a violation if its abstract state is $(\cdot, (s_E, \cdot))$, i.e., if the observer analysis that monitors the observer automaton for the specification transitions into an accepting state. The initial state is $(l_3, (s_0, \mathit{true}))$, i.e., the CFA is in its initial location l_3 and the observer automaton is in its initial state s_0. The first witness automaton (Fig. 9b) is produced by this verifier. The analysis marks the program entry in line 3 by writing an automaton transition with the source-code guard (l_3, `int main()`, l_4) and the state-space guard *true*, which is described by the label `3,enterFunction(main):` in our graphical representation [a]. The analysis detects that taking the `then`-branch in line 5 cannot lead to a violation of the specification, and thus, writes a transition to the sink state $q_{\perp_1}$, and similarly for the `else`-branch in line 20. Because the analysis does not include any information about program variables, it is unable to eliminate any infeasible program path to the function-call operations in lines 17 and 21, and thus, it writes transitions to the accepting states q_{E_1} and q_{E_2} to represent each of these violations. Consequently, the resulting witness only overapproximates the set of feasible error paths: In fact, all of the error paths that lead to the violation in line 21 are actually infeasible, and there is no restriction on the values of program variables or the number of loop unrollings, because the analysis is very imprecise.

Witness Refinement. In the next step of our example scenario, we use a verifier that runs an analysis based on an interval domain, and we give it as input the verification task and the witness from Fig. 9b, which was produced in the previous step. The Composite CPA we use for this analysis consists of the following component CPAs: A Location CPA to track the program counter, an observer analysis to track the control state of the observer automaton for the specification, a witness analysis (Automaton CPA for a witness automaton) to track the control state of the witness automaton, and an Interval CPA to track the values of variables using an interval domain. Abstract states of this Composite CPA are tuples $(l, (s, \psi_s), (q, \psi_w), e)$, where l and (s, ψ_s) again represent the component abstract states of the Location CPA and the observer analysis for the observer automaton for the specification, respectively, (q, ψ_w) is the component abstract state of the witness analysis that consists of the current control state q and state-space condition ψ_w from the witness automaton, and e is the component abstract state of the Interval CPA, that is, a mapping from the set X of program variables to intervals. Because we are consuming a violation-witness automaton, we will only consider specification violations where both the observer automaton for the specification and the witness automaton for the violation witness are in an accepting state. The initial state is $(l_3, (s_0, \mathit{true}), (q_{init}, \mathit{true}), \{n \mapsto [-\infty, \infty], v \mapsto [-\infty, \infty], s \mapsto [-\infty, \infty], i \mapsto [-\infty, \infty]\})$, i.e., the CFA is in its initial location l_3, the observer automaton for the specification is in its initial state s_0, the witness automaton is in its initial state q_{init}, and there is currently no information on variable values. From the initial program location l_3, the Location CPA only allows a transition via the CFA edge to l_4, which means that the first analyzed operation is the program entry in line 3. In the specification automaton, only the self transition on state s_0 labelled "*o/w*" matches; recall from Sect. 3.1 that such self-transitions only match if no other transitions are applicable, and that they impose no state-space restrictions. In the witness automaton, the transition from q_{init} to q_1 labeled `3,enterFunction(main):` matches. We do not gain any information on variable values. Therefore, the successor composite abstract state is $(l_4, (s_0, \mathit{true}), (q_1, \mathit{true}), \{n \mapsto [-\infty, \infty], v \mapsto [-\infty, \infty], s \mapsto [-\infty, \infty], i \mapsto [-\infty, \infty]\})$. Next, the analysis progresses via the operation on line 4, which declares the variable `n` of type `unsigned char` and initializes it via an input value by calling the function `__VERIFIER_nondet_char(void)`. Hence, the new location in the CFA is l_5, the observer automaton for the specification again takes the self-transition and stays in s_0, the witness automaton also has no matching transition other than the self-transition *o/w* at q_1 and therefore stays in q_1, and the new interval abstract state is $\{n \mapsto [0, 255], v \mapsto [-\infty, \infty], s \mapsto [-\infty, \infty], i \mapsto [-\infty, \infty]\}$. At l_5, the CFA branches. We first consider the branch to l_6. The observer automaton for the specification stays in s_0 again, but the witness automaton has a matching transition to $q_{\perp_1}$. The interval analysis detects that due to the branching condition, $n \mapsto [0, 0]$ and updates its successor component abstract state accordingly. However, since $q_{\perp_1}$ is a sink state, the witness analysis that tracks the control state of the witness automaton will not produce any further successors on this branch, so we can eliminate all corresponding program paths and need not consider them any more. We now consider the branch from l_5 to l_8. Here, the observer automaton for the specification also stays in s_0, the witness automaton stays in q_1, and the new interval abstract state is $\{n \mapsto [1, 255], v \mapsto [-\infty, \infty], s \mapsto [-\infty, \infty], i \mapsto [-\infty, \infty]\}$. After the next two operations on lines 8 and 9, which declare the variables `v` and `s` of type `unsigned char` and initializes them both to `0`, the composite abstract state is $(l_{10}, (s_0, \mathit{true}), (q_1, \mathit{true}), \{n \mapsto [1, 255], v \mapsto [0, 0], s \mapsto [0, 0], i \mapsto [-\infty, \infty]\}$. The next operation, in line 10, declares the variable `i` of type `unsigned char` and initializes it to `0`. The CFA is then in location l_{11}, which is a loop head. Therefore, the witness-automaton transition from q_1 to q_2 matches, and the composite abstract state is $(l_{11}, (s_0, \mathit{true}), (q_2, \mathit{true}), \{n \mapsto [1, 255], v \mapsto [0, 0], s \mapsto [0, 0], i \mapsto [0, 0]\}$. Next, we follow the branch from the loop head l_{11} to l_{12}, i.e., into the loop, which leads over the locations l_{12}, l_{13}, and l_{14}

and the witness-automaton state q_3 eventually back to the loop head l_{11} and the witness-automaton state q_2. We assume that the analysis is able to compute a fixed point, but at the loss of precision: When entering the loop, `i` must be lower than `n`, which is at most `255`, so `i` must be at most `254`, and since `i` is incremented within the loop, it must be between `1` and `255` at the end of each loop iteration. The most precise single abstract state in our current composite abstract domain that covers all reached states at the loop head l_{11}, however, is $(l_{11}, (s_0, \mathit{true}), (q_2, \mathit{true}), \{n \mapsto [1, 255], v \mapsto [0, 255], s \mapsto [0, 255], i \mapsto [0, 255]\}$. After reaching this fixed point for the loop, we continue with the branch from l_{11} to l_{16}, which matches the source-code guard `11,else` on the witness-automaton transition from q_2 to q_4. At this point, we encounter another branching. We first take the branch from l_{16} to l_{17}, which matches the source-code guard `16,then` on the witness-automaton transition from q_4 to q_{E_1}, but since the specification automaton still stays in s_0, which is not an accepting state, we continue to the following operation, which is the function call to `__VERIFIER_error(void)` on line 17. This operation matches the source-code guard `enterFunction(__VERIFIER_error)` on the observer-automaton transition from s_0 to the accepting state s_E. Because the witness automaton stays in the accepting state q_{E_1}, the analysis has now found a program path to a specification violation that is described by the input witness, and it writes a transition to the corresponding accepting control state q_{E_1} into its output witness (Fig. 9c). We now follow the branch from l_{16} to l_{20}. For this operation, we compute the successor state $(l_{20}, (s_0, \mathit{true}), (q_5, \mathit{true}), \{n \mapsto [1, 255], v \mapsto [0, 255], s \mapsto [0, 255], i \mapsto [0, 255]\})$. We then encounter another branching. When attempting to compute the successor abstract state via the branch from l_{20} to l_{21}, the interval component will detect that all program paths along this branch are infeasible. Therefore, no successor abstract state for this branch is computed. Along the other branch from l_{20} to l_{24}, the source-code guard `20,else` on the witness-automaton transition to $q_{\perp_2}$ matches, so that the analysis does not continue along this branch either and the state-space exploration is complete. Because the analysis did not encounter any violation states via the `else`-branch in line 16, it writes a corresponding transition to a sink state into its output witness. The new witness (Fig. 9c) is more precise than the input witness (Fig. 9b), as it does not contain the infeasible error paths to line 21 and puts restrictions on variable values, but it is still an overapproximation of the set of feasible error paths. For example, it contains infeasible error paths that never enter the loop and therefore cannot trigger the overflow that makes the violation in line 17 reachable.

Execution-Based Witness Validation. In the third step of our example scenario, the violation witness from Fig. 9c is used to restrict a test-case generator [26] to derive a specific test vector for the program path to the call to `__VERIFIER_error(void)` in line 17. The test vector is derived by extracting a satisfying assignment of the formula that represents the program path to l_{18}. The third witness automaton (Fig. 9d) represents the result of the test-case generation, i.e., a test vector, which contains all the input data necessary to execute a test of the program that triggers the described violation of the specification. This witness precisely represents exactly one feasible error path, and is therefore an underapproximation of the set of feasible error paths, because there are also other paths that would lead to a violation, for example with more than two loop iterations or with a different pair of summands.

[a]While the token `enterFunction(main)` would already be sufficient to unambiguously describe the source-code guard, we always add the line number for the reader's convenience.

Dynamic Validation. We call CPA-witness2test and FShell-witness2test *dynamic* validators, because they perform only very light-weight static analysis to extract a test vector from a violation witness, and then compile, link, and execute the program with a corresponding test harness to dynamically

```c
struct _IO_FILE;
typedef struct _IO_FILE FILE;
extern struct _IO_FILE *stderr;
extern int fprintf(FILE *__restrict __stream, const char *__restrict __format,
↪ ...);
extern void exit(int __status) __attribute__ ((__noreturn__));
void __VERIFIER_error() {
  fprintf(stderr, "cpa_witness2test:␣violation\n");
  exit(107);
}
unsigned char __VERIFIER_nondet_char() {
  static unsigned int test_vector_index = 0;
  unsigned char retval;
  switch (test_vector_index) {
    case 0: retval = (unsigned char)2; break;
    case 1: retval = (unsigned char)224; break;
    case 2: retval = (unsigned char)63; break;
  }
  ++test_vector_index;
  return retval;
}
```

Fig. 10. Test harness generated from the witness of Fig. 9d for the C program of Fig. 9a

observe whether the specification is actually violated during the execution. The advantages of dynamic validators are that (1) they can be much more efficient than static validators, because they do not require expensive model-checking techniques, (2) they can be more precise than static validators, because a violation that can be observed during an actual execution of a program undeniably exists, (3) an executable produced by a dynamic validator can be used by developers to analyze a bug by applying standard tools they already know and are well-trained in, such as debuggers, and (4) a test harness produced by a dynamic validator can directly be used by developers to improve their test suite and prevent regressions once the bug is fixed. There are, however, also some disadvantages: A dynamic validator requires as input a witness that represents a test vector, i.e., a witness that specifies concrete value assignments for all program inputs, which may not be available from all verifiers. To obtain a suitable witness, it is possible to first apply witness refinement to the original witness, but, because witness refinement uses the expensive techniques of static validators, this solution negates the first advantage of dynamic validators over static validators (i.e., their efficiency). A second disadvantage of dynamic validators is that they require a concrete and secure execution environment[7] that matches the target environment of the analyzed system, whereas static validators can, conceptually, also be used in any (unrelated) other execution environment. Lastly, dynamic validators can only confirm a violation if the time required to execute the program and trigger the bug is finite (and reasonably brief). If, for example, the specification is that the program must always terminate, a validator for a violation would need to confirm that there is a path that does not terminate; this cannot be observed from a finite execution.

Example 8 (Execution-Based Validation). We now demonstrate execution-based validation of verification results as it would be performed by the validator CPA-WITNESS2TEST when applied to the verification task composed of the program from Fig. 9a and the specification from Fig. 2, and the third and most precise witness from our previous example shown in Fig. 9d. To extract the input values from the witness, match them to the input functions of the program, and

[7]Test-suite validators (such as TESTCOV [41]) can be used for safe and secure execution of tests.

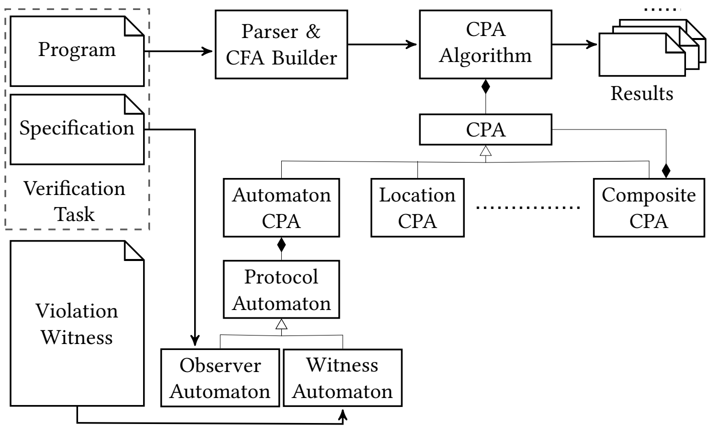


Fig. 11. Architecture of the violation-witness-based result validator implemented in CPAchecker

generate the test harness depicted in Fig. 10, the validator first runs a light-weight program analysis configured as a Composite CPA composed of the following components: a Location CPA that tracks the program counter, an observer analysis that tracks the control state of the observer automaton for the specification, and a witness analysis that tracks the control state of the witness automaton for the violation witness. The analysis traces the program paths that are described by the witness automaton and matches the values of input variables on these paths to the corresponding input functions of the program in the correct order. In the example, the automaton specifies that in line 4, the value assigned to the variable `n` (which controls how many further input values will be read) should be `2`; that in the first loop iteration in line 12, the value assigned to the variable `v` should be `224`; and that in the second loop iteration in line 12, the value assigned to the variable `v` should be `63`. Consequently, the validator produces the test harness listed in Fig. 10, which contains an implementation of the input function `__VERIFIER_nondet_char(void)` that returns exactly those values in that order, and an implementation of the function `__VERIFIER_error(void)` that, if called, allows the validator to detect the specification violation through a custom program output. In the next step, the validator compiles and links the source code of the C program and the produced test harness, and executes the resulting program. As expected, the validator can observe the specification violation during execution, because the sum of `224` and `63` is `287`, which exceeds the value range of the type `unsigned char` of variable `s` and therefore wraps around to the value `31`. Because `31` is less than the last input value `63`, the function `__VERIFIER_error()` is called at line 17. The validator detects this function call and confirms the verification result.

4.2.2 *Tool Implementations.*

We implemented violation-witness-based result validation in the two static validators CPAchecker and UAutomizer, and in the two dynamic validators CPA-witness2test and FShell-witness2test.

Static Violation-Witness-Based Result Validation with CPAchecker. Figure 11 shows a section of the architecture of CPAchecker that implements violation-witness-based result validation. The left side of the figure shows the inputs, consisting of the verification task (i.e., program and specification) and the violation witness. The program is parsed and converted into a CFA, the specification into an observer automaton, and the violation witness into a witness automaton. Then, the CPA algorithm is run with a composite program analysis that is composed of at least a Location CPA and two Automaton CPAs, one for the observer automaton for the specification and one for the witness automaton for the violation witness. As mentioned in Sect. 4.1.1, further CPAs are available and

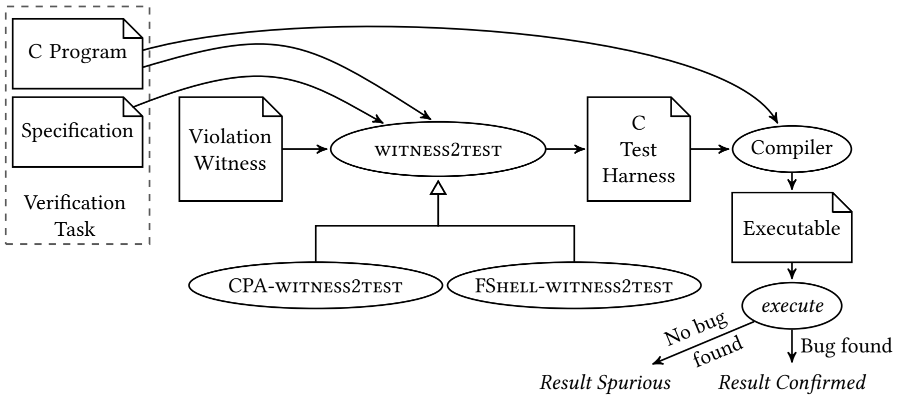


Fig. 12. Flow of violation-witness-based result validation with dynamic validators (witness2test)

can optionally be added to the composition to enhance the capabilities of the program analysis; in our evaluation (cf. Sect. 6), we add predicate analysis [28] and explicit-state model checking [42].

After the CPA algorithm completes, the computed results are delivered. This may either be a simple confirmation (or refutation) of the validated result, or it may be a refined violation witness if the validated result was confirmed and the validation process was able to add information to an imprecise input witness. The witness is confirmed if the observer automaton and the witness automaton *both* reached one of their accepting contol states.

Static Violation-Witness-Based Result Validation with UAutomizer. While we formalized witness-based result validation using the concept of configurable program analysis, other approaches are also applicable: UAutomizer performs the validation in two steps. In the first step, a new CFA is constructed that represents those paths of the original CFA that comply with the source-code guards of the witness automaton, i.e., the new CFA is constructed as a product of the original CFA and the witness automaton. The states of this product automaton are pairs (l, q), where l is a location of the original CFA and q is a control state of the witness automaton. The product contains a transition from (l, q) to (l', q') labeled with op if

- (l, op, l') is a CFA edge and
- $q \xrightarrow{(S,\psi)} q'$ is a transition in the witness automaton such that $(l, op, l') \in S$.

In the second step, UAutomizer verifies if the resulting CFA satisfies the specification using its automata-theoretic verification approach [84]. The witness is confirmed if a violation of the specification is found, that is, the observer automaton for the specification reached an accepting control state and the new CFA also reached an accepting control state.

Dynamic Violation-Witness-Based Result Validation with CPA-witness2test. Figure 12 shows the witness2test-workflow for violation-witness-based result validation with dynamic validators: The validator receives as input the verification task and the violation witness produced by a verifier, and synthesizes from these inputs a test harness for the program. This test harness and the program source code are compiled and linked with a C compiler to produce an executable program, which is then executed. Assuming that the witness represents a precise test vector and the validator correctly translates the witness into a test harness, if the validator observes a violation, then the bug found by the verifier and described by the witness is realizable and the result is therefore confirmed; otherwise, it is refuted. Because an executable program that triggers an actual bug is always available after a successful validation, the developer can immediately start debugging, for example by running the executable with a debugger like GDB.

For CPA-witness2test, the step of extracting the test vector from the verification task and witness is conceptually similar to the validation performed by static validators, except that the static analysis is used only to assign the input values from the witness to the correct input functions of the program, not to perform any further semantic reasoning about the program. In fact, CPA-witness2test uses also the architecture displayed in Fig. 11 for the step of matching the input values from the witness to the input functions of the program. Unlike the static validators, however, it does not add further CPAs and its result in this step is the test harness.

Dynamic Violation-Witness-Based Result Validation with FShell-witness2test. The key design principle of FShell-witness2test, on the other hand, is independence from existing verification infrastructure: the results of FShell-witness2test are by design unbiased towards any existing software-analysis framework. Consequently, FShell-witness2test is another example that shows that while we formalize witness-based result validation using the CPA concept, implementations that follow other paradigms are also applicable in practice. The architecture of FShell-witness2test consists of two major parts: (1) a Python-based processor of the violation witness and the program source code, using `pycparser`,[8] to generate a test vector in a format compatible with FShell [89] (hence the name of the validator), and (2) a Perl script to convert such a test vector into a test harness that can be compiled and linked with the input program. For a given violation witness and verification task, FShell-witness2test first parses the specification to determine the expected type of violation. The witness and the C program are then handed to the Python-based processor. Because `pycparser` cannot handle various GCC extensions, input programs are preprocessed and sanitized by performing text replacement and removal. FShell-witness2test then obtains the abstract syntax tree and iterates over its nodes to gather data types and source locations of input-value assignments. Finally, FShell-witness2test builds a linear sequence of states from the witness automaton. Traversing this sequence, any match of line numbers against the input-value assignments triggers an attempt to extract values from assumptions in the witness. If the assumption represents a precise value assignment, an input value is recorded.

4.3 Result Validation Based on Correctness Witnesses

Correctness witnesses are verification witnesses that represent the artifacts of a proof that a program satisfies a specification, i.e., invariants for certain program locations that are intended to help reconstruct a correctness proof (Sect. 3.2.3).

4.3.1 Principles. The program analysis of a correctness-witness-based result validator checks if the given invariants indeed hold at their corresponding abstract program states; validation of a correctness witness fails if the validator refutes the invariant φ for an abstract program state or if it detects a violation of the specification, i.e., a feasible error path.

There are only two differences between violation-witness validation and correctness-witness validation:

- Violation-witness-based result validation uses assumptions at the witness automaton's transitions to constrain the state space; a correctness witness does not constrain the state space but contains at each control state in the witness automaton a state invariant.
- Violation-witness-based result validation attempts to replay an error path through the program, while correctness-witness-based result validation tries to replay the correctness proof: after confirming a witness invariant, it may use it as an auxiliary lemma to prove the correctness of the program or further witness invariants.

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

4.3.2 ***Tool Implementations.*** Currently, two implementations of correctness-witness-based result validators exist. We describe the two different strategies employed by CPAchecker and UAutomizer (out of the many possible strategies to implement a validator), which are implemented in two different verification frameworks to demonstrate the potential and flexibility of the approach.

CPAchecker's Correctness-Witness-Based Result Validator. Like the CPAchecker-based verifier from Sect. 4.1.1, the CPAchecker-based validator for correctness witnesses uses the KI⟲KI technique for k-induction. In a preparatory step, the invariants are extracted from the correctness witness and mapped to their corresponding program locations. By design, a witness may be imprecise, therefore it is possible that an invariant is mapped to several program locations. The invariant generator then uses these invariants as candidate invariants, and, if it is able to prove inductiveness of such a candidate invariant, it supplies it as an invariant to the main k-induction procedure. If, on the other hand, the invariant generator is able to refute a candidate at all program locations described by its corresponding state in the witness automaton, the validation fails, i.e., refutes the result.

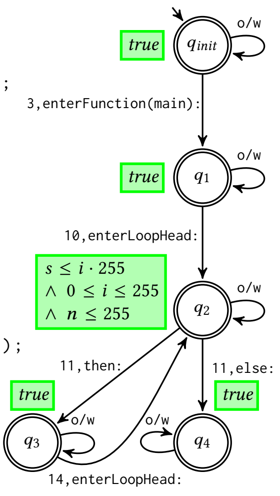
One of the advantages of using k-induction for correctness-witness-based result validation is that for non-trivial software-verification tasks, k-induction is known to perform well only if it is supplied with the necessary auxiliary invariants [27, 99]. By design, all techniques that are implemented in CPAchecker to generate its own auxiliary invariants (e.g., from data-flow analysis) are turned off for the validation. Consequently, the validator's success in confirming a proof result depends on the quality of the invariants given by the witness.

Within each iteration of the k-induction procedure of the KI⟲KI technique's invariant generator, CPAchecker will try to refute each invariant provided by the witness by finding a counterexample of the current length k before trying to prove its correctness. Hence, CPAchecker is guaranteed to find incorrect invariants with counterexamples that are at most as long as the value of k required to prove that the program conforms to its specification, and it is also guaranteed to only use supplied invariants that it can prove to be correct. CPAchecker does not guarantee, however, that it will detect incorrect witness-supplied invariants if the length of their shortest counterexample exceeds the value of k required to prove that the program itself is correct. This is a design decision of the implementation, not a limitation of the concept of correctness witnesses or the CPAchecker framework. To instead exhaustively confirm or refute all provided invariants, CPAchecker could be changed to simply defer checking the correctness of the program until the KI⟲KI invariant generator has processed all invariants. The reasoning for the design decision without exhaustive checking was to not discourage developers of verifiers from producing invariants that k-induction might struggle with and cause exhaustive checks to time out. Moreover, if proving the correctness of a program requires an auxiliary invariant and the witness provides a correct one, the witness can already be considered useful, even if not all of its contents is checked exhaustively. For use cases where exhaustive proof or refutation of all invariants is desired, an alternative implementation is provided by UAutomizer.

Ultimate Automizer's Correctness-Witness-Based Result Validator. To validate a proof result, UAutomizer verifies the given program and considers each invariant provided by the correctness witness as an additional specification. Each of the specifications (the additional specifications from the invariants and the original specifications) is checked in the order of their occurrence in the program, and if correct, can be assumed while checking specifications occurring later in the program. UAutomizer confirms the result if the original specification and all specifications derived from witness invariants hold. If one specification cannot be confirmed, the validation fails (result refuted). In case we do not want to validate the witness as a whole but would like to point out incorrect invariants individually, we can check each specification individually without assuming validity for any other specification.

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

(a) Safe program (from Fig. 1a)



(b) Witness automaton

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

(c) Unsafe program (from Fig. 9a)

Fig. 13. Example C programs (a and c) and a potential correctness witness (b), with the only difference between the two programs (line 9) highlighted

Converting a witness invariant into an additional specification is implemented as follows. First, an observer analysis matches the program CFA against the witness to obtain a partial map f from program locations to witness invariants. In a second step, the CFA is modified as follows. For each location l for which the mapping f is defined, UAutomizer

- adds a new location l',
- adds a new edge $(l, op_{f(l)}, l')$ where $op_{f(l)}$ is the assume operation that assumes the invariant $f(l)$ that was mapped to l,
- adds a new edge $(l, op_{\neg f(l)}, l_{\text{err}})$, where $op_{\neg f(l)}$ is the assume operation that assumes the negation of the invariant $f(l)$ and l_{err} is a location whose reachability is forbidden by the original specification, and
- replaces each outgoing edge of the form (l, op, l'') by an edge (l', op, l'').

The resulting CFA is verified as described in Sect. 4.1.2.

Example 9 (Correctness Witnesses). We illustrate the idea of correctness-witness validation using two short C programs listed in Fig. 13a (taken from Fig. 1a) and Fig. 13c (taken from Fig. 9a), an example correctness-witness automaton shown in Fig. 13b, and the specification from Fig. 2, which forbids reachable calls to the function `__VERIFIER_error(void)`. The first of the two C programs (Fig. 13a) differs from the second C program (Fig. 13c) only in one line: While variable `s` is declared with type `unsigned int` in line 9 of the first program, it is declared with type `unsigned char` in line 9 of the second program. As a result, the first program satisfies the specification, while the second program violates it, because `s` is susceptible to arithmetic overflows (cf. Example 7).

One way to prove that neither of the calls to the function `__VERIFIER_error()` in line 17 and line 21 is reachable in the first program would be to find an upper bound for the number of loop iterations and then unroll the loop, performing a bounded model check [50]. While this would still be feasible in our example due to the small types and simple structure that we chose for ease of presentation, it would already be expensive; for larger loop bounds, this strategy becomes infeasible, and for unknown loop bounds, it is impossible. Another —unbounded— way to prove that the program is safe is to prove that $s \leq 65025$ is an invariant of the loop from lines 11 to 15. This invariant is not inductive for the loop, however: It holds trivially before the loop, where $s = 0$, but the loop body does not guarantee that the invariant is preserved. Even strengthening this invariant to $s \leq 65025 \wedge i \leq 255$ (which should be simple for any verifier that understands C types and knows that in our target architecture, type `unsigned char` is 8 bits wide) is not inductive. The predicate $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$, on the other hand, is an inductive invariant of the loop, but finding such an invariant is usually difficult, and depends on the verification strategy: it is therefore the critical step in solving this verification task.

A verifier that successfully proves the safety property for the program may then export a correctness witness. If the correctness witness contains the invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$, a witness validator using the witness should be able to easily confirm the proof. Figure 13b displays a graphical representation of such a correctness witness. The automaton starts in an initial control state q_{init}. The witness assigns the invariant *true*. It is allowed to proceed to state q_1 if the control flow enters the `main` function of the program. As long as this transition is not possible, the automaton remains in state q_{init} via the self-transition 'otherwise' (*o/w*). From q_1, the automaton is allowed to proceed to q_2 if the control-flow enters the loop head; otherwise, it remains in q_1 via the self-transition *o/w*. From q_2, the automaton can proceed to control state q_3 if the condition of the `while` loop in line 11 is true (the `then`-case), or to state q_4 if the condition in line 11 is false (the `else`-case). As long as none of these transitions are possible, the automaton remains in control state q_2 via the self-transition *o/w*. The automaton proceeds back from state q_3 to q_2 after the program operation in line 14; as long as this is not possible, the automaton will stay in q_3 via its self-transition *o/w*. If the automaton is in control state q_4, it will stay there forever.[a] Control states q_{init}, q_1, q_3, and q_4 contain the trivial state invariant *true*. Control state q_2 specifies the invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$. Because state q_2 describes the loop head, a validator is able to prove (for example by induction) that the invariant holds at this program location, and can then use the invariant to prove the correctness of the program, thus validating the proof result.

If the invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$ is removed from the witness for state q_2, the witness is still valid (because *true* is an invariant). However, the k-induction-based validator will no longer confirm the proof because it lacks the information that is required to prove the correctness of the program, and it is not allowed to synthesize the required information itself. This is a design choice, in order to not confirm witnesses that are extremely weak (e.g., *true* everywhere).

Due to the structural similarity between the first program in Fig. 13a and the second program in Fig. 13c, the witness in Fig. 13b can also be matched with the second (unsafe) program. In this case, however, the loop invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$ does not imply that $s \geq v$, because the invariant is no longer sufficient to preclude overflows during the addition in line 13. In fact, if we conjoined $s \geq v$ to our invariant, it would still be a valid loop invariant in Fig. 13a but not in Fig. 13c, where an overflow would be a counterexample to the inductiveness of $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255 \wedge s \geq v$. Hence, a validator will not be able to prove the correctness of the program using the loop invariant $s \leq i \cdot 255 \wedge 0 \leq i \leq 255 \wedge n \leq 255$. Because each correctness-witness-based validation of a proof result also implicitly uses the safety property as an invariant, the validator can reject the witness by finding a feasible error path to line 17 as a counterexample to the specification, such as the one from Fig. 9d. The strength of the invariants determines the quality of the witnesses, but no particular strength is required. This example shows that correctness-witness-based validation can be more efficient than verification because it might be easier to (re-) verify that invariants indeed hold, while the verification needs to come up with the invariants. The task of finding useful invariants is in general considered one of the key challenges in software verification. Generalizing this approach allows for a lot of flexibility, because the more helpful the candidate invariants are, the less work has to be performed by the validator.

[a]The rest of the exploration does not matter for the witness, because the sole purpose of the witness is to attach the invariant at the right program location.

5 EXCHANGE-FORMAT SPECIFICATION

To store witness automata and exchange them across different verification tools and validators, we define an exchange format. Because automata are graphs, we use the existing graph format GraphML [51], which defines XML elements for edges (used in our format to model protocol-automaton transitions) and nodes (to model protocol-automaton control states).

The root element of a protocol-automaton GraphML document is the element `graphml`. The graph that models the protocol automaton is represented by the element `graph`, which is a child element of the root element `graphml`. We require that there is exactly one such `graph` element in the document. We model a control state of a protocol automaton using a `node` element. Each `node` element is a child element of the `graph` element and must specify a unique identifier (within the graph) for the control state using the attribute `id`. Analogously, we model a protocol-automaton transition using an `edge` element. Each `edge` element is a child element of the `graph` element and has the attributes `source` and `target`, both of which refer to `node` elements via their `id`s. Additional data can be attached to individual nodes and edges, and the graph itself, by adding `data` elements as child elements. The content of a `data` element is its value; each data element must specify its meaning via a `key` attribute. Each `key` that is used in the document must be defined using a `key` element as a child element of the root element `graphml`. A `key` element must define the name of the key (using the attribute `attr.name`), the type of the values of `data` elements with this key (using the attribute `attr.type`), whether data elements with this key are used on the `graph`, or on `node` or `edge` elements (using the attribute `for`), and a unique identifier for the key (using the attribute `id`. Valid values for the `attr.type` attribute in GraphML are `boolean`, `int`, `long`, `float`, `double`, and `string`. For protocol automata, we use the type `boolean` for boolean values, `int` for integer values, and `string` for other values. The `key` attribute of a `data` element refers to the value of the `id` attribute of the corresponding `key` element, not its name. A default value can be defined for each key as the content of a `default` element that is added to the desired `key` element as a child element.

Keys for `graph` Elements. The following keys are defined for `data` elements that are used to add information that concerns the witness as a whole, i.e., for `data` elements that are direct children of the `graph` element:

- `witness-type` is used to specify the witness type. A correctness witness is identified by the value `correctness_witness`, a violation witness is identified by the value `violation_witness`.

- `sourcecodelang` is used to specify the name of the programming language, for example `C`.
- `producer` is used to specify the name of the tool that produced the witness automaton, for example `CPAchecker 1.6.8`.
- `specification` is used to provide a textual representation of the specification of the verification task. The format of this representation is user-defined. In SV-COMP [12], the text `CHECK( init(main()), LTL(G ! call(__VERIFIER_error())))` is used to represent the specification from Fig. 2.
- `programfile` is used to record the URI or file-system path to the source code, e.g., `loop-acceleration/multivar_true-unreach-call1_true-termination.i`. This key is intended for documentation purposes; a validator is not required to be able to access the specified file location, because the source code is explicitly provided to the validator as input. Hence, the validity of the witness must not depend on the availability of the source code at the location specified by the value of this key in the execution environment of the validation.
- `programhash` is used to record the SHA-256 hash value of the verified program.
- `architecture` is used to provide a textual representation of the machine architecture assumed for the verification task. This textual representation is user-defined. We propose to use the identifiers `32bit` and `64bit` to distinguish between 32-bit systems and 64-bit systems.
- `creationtime` is used to specify the date and time the witness was created in ISO 8601 format. The date must contain the year, the month, and the day, separated by dashes (“`-`”). The date is separated from the time using the capital letter “`T`”. The time must be given in hours, minutes, and seconds, separated by colons (“`:`”). If the timestamp is in UTC time, it ends with a “`Z`”. If the timestamp is not given in UTC time, a positive (“`+`”) or negative (“`-`”) time offset consisting of hours and minutes separated by a colon (“`:`”) can be appended. Example: `2016-12-24T13:15:32+02:00`.

We require that values for all keys listed above are provided. The value of the `attr.type` attribute of all `key` elements corresponding to these keys is `string`.

Keys for `node` Elements. The following keys are defined for `node` elements, which represent control states in witness automata:

- `entry` is used to mark a `node` as an entry node. An entry node represents the initial control state of the witness automaton. We require that exactly one initial control state is defined per document. The `attr.type` attribute of this key is `boolean`. The default value is `false`.
- `sink` is used to mark a `node` as a sink node. A sink node represents a sink control state in the automaton. Sink states are not allowed in correctness-witness automata (Tables 1 and 4). The `attr.type` attribute of this key is `boolean`. The default value is `false`.
- `violation` is used to mark a control state as a violation state, i.e., as a state that represents a specification violation. Violation control states are not allowed in the syntax for correctness witnesses, because all control states are implicitly accepting states (Tables 1 and 4). The `attr.type` attribute of this key is `boolean`. The default value is `false`.
- `invariant` is used to specify an invariant for a control state. The value of a `data` element with this key must be an expression that evaluates to a value of the equivalent of a boolean type in the programming language of the verification task, e.g., for C programs, a C expression that evaluates to a value of the C type `int` (used as boolean). The expression may consist of conjunctions or disjunctions, but not function calls. Local variables that have the same name as global variables or local variables of other functions can be qualified by using a `data` element with the `invariant.scope` key. Invariants are not allowed in violation-witness automata (Tables 1 and 4). The `attr.type` attribute of this key is `string`. All variables used in the expression must

Table 4. Keys for `node` elements allowed in witnesses

Key	Violation witness	Correctness witness
`entry`	✓	✓
`sink`	✓	
`violation`	✓	
`invariant`		✓
`invariant.scope`		✓
`cyclehead`	✓	

appear in the program source code. If a control state does not have a `data` element with this key, a consumer shall consider the state invariant to be *true*.

- `invariant.scope` is used to qualify variables with ambiguous names in a state invariant by specifying a function name: The witness consumer must map the variables in the given invariant to the variables in the source code. Due to scopes in many programming languages, such as C, there may ambiguously named variables in different scopes. The consumer first has to look for a variable with a matching name in the scope of the function with the name specified via a `data` element with the `invariant.scope` key before checking the global scope. This key always applies to the invariant as a whole, i.e., it is not possible to specify an invariant over local variables of different functions. In existing implementations, there is currently no support for different variables with the same name within different scopes of the same function. Invariant scopes are not allowed in violation-witness automata (Tables 1 and 4). The `attr.type` attribute of this key is `string`.
- `cyclehead` is used to mark a state that connects stem and loop in a violation witness for termination, i.e., it marks the separation of *stem* and *loop* of a non-termination *lasso* [82]. A state with this annotation should be reachable from every non-sink state in the loop. At least one such state is required in a violation witness for termination properties. In reachability witnesses, this annotation is not allowed. The `attr.type` attribute of this key is `boolean`. The default value is `false`.

In general, it is not required to annotate a `node` element with `data` elements, except (a) that one `node` must be specified to represent the initial state of the automaton using a `data` element with the `entry` key, (b) that in a violation-witness automaton, there should be at least one control state that is marked as a violation state using a `data` element with the `violation` key, and (c) that in a violation witness for a termination specification, there should be at least one control state that uses a `data` element with the `cyclehead` key.

Keys for `edge` Elements. The following keys are defined for `edge` elements, which represent transitions in witness automata:

- `assumption` is used to specify a state-space guard for a transition. The value of a `data` element with this key must be an expression that evaluates to a value of the equivalent of a boolean type in the programming language of the verification task, e.g. for C programs, a C expression that evaluates to a value of the C type `int` (used as boolean). The expression may consist of conjunctions or disjunctions, but not function calls. Local variables that have the same name as global variables or local variables of other functions can be qualified by using a `data` element with the `assumption.scope` key. All variables used in the expression must appear in the program source code, with the exception of the variable `\result`, which represents the return value of a function identified by the `data` element with the key `assumption.resultfunction` after a function-return operation on a CFA edge matched by this transition. If the `\result` variable is used, the name of the corresponding function must be provided using a `data` element with the `assumption.resultfunction` key. If a transition does not have a `data` element with the `assumption` key, a consumer shall assume that the state-space guard of this transition is *true*. In correctness witnesses, for each state and each source-code guard, the disjunction of all state-space guards leaving that state via a transition matched by that source-code guard must be *true*, i.e., while state-space guards can be used to split the state space in correctness witnesses, they may not be used to restrict it (Table 1). The `attr.type` attribute of this key is `string`.

- `assumption.scope` is used to qualify variables with ambiguous names in a state-space guard by specifying a function name: The witness consumer must map the variables in the given invariant to the variables in the source code. Due to scopes in many programming languages, such as C, there may ambiguously named variables in different scopes. The consumer first has to look for a variable with a matching name in the scope of the function with the name specified via a `data` element with the `assumption.scope` key before checking the global scope. This key always applies to the state-space guard as a whole, i.e., it is not possible to specify a state-space guard over local variables of different functions. In existing implementations, there is currently no support for different variables with the same name within different scopes of the same function. This key is not allowed in correctness witnesses (Table 1). The `attr.type` attribute of this key is `string`.
- `assumption.resultfunction` is used to specify the function of the `\result` variable used in a state-space guard of the same transition, meaning that `\result` represents the return value of the given function. This key applies to the state-space guard as a whole, it is therefore not possible to refer to multiple function-return values within the same transition. If the `\result` variable is used, a `data` element with this key must be used in the same transition, otherwise it is superfluous. This key is not allowed in correctness witnesses (Table 1). The `attr.type` attribute of this key is `string`.
- `control` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to assume operations of the CFA. Valid values for `data` elements with this key are `condition-true` and `condition-false`, where `condition-true` specifies the branch where the assume condition evaluates to *true*, i.e., the `then` branch, and `condition-false` specifies the branch where the assume condition evaluates to *false*, i.e., the `else` branch. The `attr.type` attribute of this key is `string`.
- `startline` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations on specific lines in the source code. Any line number of the source code is a valid value for `data` elements with this key. A `startline` refers to the line number on which an operation of a CFA edge begins. The `attr.type` attribute of this key is `int`.
- `endline` is similar to the `startline` key, except that it refers to the line number on which an operation of a CFA edge ends.
- `startoffset` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations between specific character offsets in the source code, where the term character offset refers to the total number of characters from the beginning of a source-code file up to the beginning of some intended statement or expression. Any character offset between the beginning and end of a source-code file is a valid value for `data` elements with this key. While on the one hand, usage of `data` elements with this key allows the witness to convey very precise location information, on the other hand, this information is susceptible to even minor changes in the source code. If this is not desired, usage of `data` elements with this key should be omitted by the producer, or, if that is not an

option, it can be removed during a post-processing step, provided that enough other source-code guards are present to make matching the witness against the source code feasible. A third option would be to recompute the offset values for the changed source code using a diff tool. The `attr.type` attribute of this key is `int`.

- `endoffset` is similar to the `startoffset` key, except that it refers to the character offset at the end of an operation.
- `enterLoopHead` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations on CFA edges where the successor is a loop head. For our format specification, any CFA node that (1) is part of a loop in the CFA, (2) has an entering CFA edge where the predecessor node is not in the loop, and (3) has a leaving CFA edge where the successor node is not in the loop, qualifies as a loop head. Note, however, that depending on the programming language of the verification task, the loop head may be ambiguous. For example, in C it is possible to use `goto` statements to construct arbitrarily complex loops with many CFA nodes that match the definition above. Conversely, there could also be loops without any loop head matching this definition, in which case the key may not be used. The `attr.type` attribute of this key is `boolean` and its default value is `false`.
- `enterFunction` is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to function-call operations where the name of the called function matches the specified value. A witness consumer may also use this key to track a stack of function calls and use this information to qualify ambiguously named variables in state-space guards or state invariants in the absence of explicitly specified scopes via the `assumption.scope` or `invariant.scope` keys. The `attr.type` attribute of this key is `string`.
- `returnFromFunction` is the counterpart of the `enterFunction` key, i.e., it is used as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to function-return operations where the name of the function that is being returned from matches the specified value. Analogously to `enterFunction`, a witness consumer may use this key to track a stack of function calls. The `attr.type` attribute of this key is `string`.
- `threadId` is used in the analysis of concurrent programs[9] as part of the source-code guard of a transition and represents the currently active thread for the transition. The value of `data` elements with this key must uniquely identify an active (i.e., created but not yet destroyed) thread within each run through the automaton, meaning that if two different threads share the same identifier, they must either (a) be on different automaton runs or (b) at each step of each automaton run at most one of them may be active. If a transition has `data` elements where one specifies a `threadId` and another one uses the `createThread` key, the `threadId` refers to the thread that creates the new thread, not the created thread. The `attr.type` attribute of this key is `string`.
- `createThread` is used in the analysis of concurrent programs as part of the source-code guard of a transition and restricts the set of CFA edges matched by the source-code guard to operations where a new thread is created. The value of `data` elements with this key is an identifier for the new thread. Any string may be used as an identifier, provided that it uniquely identifies an active thread in each automaton run. The initial function of the created thread must be provided in a subsequent automaton transition using the `enterFunction` key, except for the main thread of the program, where the same (initial) transition may be used, because at that point, no other thread exists yet. Subsequently, a thread is assumed to be terminated once its callstack is empty again, which is achieved by using a corresponding `returnFromFunction` value. The `attr.type` attribute of this key is `string`.

[9]For more details on witnesses for concurrent programs, we refer the reader to the literature [29].

Table 5. Keys for `edge` elements allowed in witnesses

Key	Violation witness	Correctness witness
`assumption`	✓	✓
`assumption.scope`	✓	✓
`assumption.resultfunction`	✓	✓
`control`	✓	✓
`startline`	✓	✓
`endline`	✓	✓
`startoffset`	✓	✓
`endofset`	✓	✓
`enterLoopHead`	✓	✓
`enterFunction`	✓	✓
`returnFromFunction`	✓	✓
`threadId`	✓	✓
`createThread`	✓	✓

In general, it is not required to annotate an `edge` element with `data` elements, but in practice, there is rarely any value in having a completely unrestricted transition in a protocol automaton. Note that the *o*/*w*-transitions that we defined in Sect. 3.1.2 are implicit, i.e., they do not appear in the exchange format as explicit `edge` elements but are automatically synthesized by the consumer.

The format specification, including a list of keys, is maintained in a GitHub project.[10] For termination witnesses, the project contains a dedicated section.[11] An open-source witness linter for checking the well-formedness of a witness is also available [12] and has been used in SV-COMP 2021 [18].

6 EXPERIMENTAL EVALUATION

To demonstrate the applicability of our approach, we performed a large number of experiments. The experimental work flow consists of running (1) a verifier, which produces a verification witness for the obtained result, and (2) a validator, which uses the verification witness to validate the result that the verifier obtained.

6.1 Experiment Goals

In the previous section, we defined an exchange format for machine-readable witnesses, in order to enable different verifiers to document their verification results in such a way that other tools can work with those verification results. Next, we perform an experimental study to support the following claims:

Claim 1 (Consistency within the Same Framework): Most of the witnesses produced by a verifier based on a certain framework can be validated by a validator based on the same framework. If the claim does not hold, then there is an inconsistency in the communication of the verification facts via the witnesses.

Claim 2 (Validation across Frameworks): The witnesses produced by a verifier based on one framework can be understood by a witness validator of a different framework.

[10]https://github.com/sosy-lab/sv-witnesses
[11]https://github.com/sosy-lab/sv-witnesses/tree/svcomp22/termination
[12]https://github.com/sosy-lab/sv-witnesses/tree/svcomp22/lint

Table 6. Categories supported by witness validators

Category	Witness type	CPAchecker	CPA-witness2test	FShell-witness2test	UAutomizer
Concurrency	Violation	✓			
	Correctness				
MemSafety	Violation	✓	✓	✓	✓
	Correctness				✓
Overflows	Violation	✓	✓	✓	✓
	Correctness				✓
ReachSafety	Violation	✓	✓	✓	✓
	Correctness	✓			✓
Termination	Violation	✓			✓
	Correctness				

Claim 3 (Effectiveness and Efficiency of Validation Depends on Witness Contents): There are verification tasks for which a verifier can produce witnesses such that the validation uses less resources to validate the result based on the witness, than the verifier used to solve the verification task.

We evaluate these claims separately for violation witnesses and correctness witnesses. Furthermore, we distinguish between five different categories (which correspond to five different specifications) of verification tasks, because not all verifiers and validators support all categories (see Table 6).

6.2 Benchmark Set

Our benchmark set consists of all 10 521 verification tasks from all categories of SV-COMP 2019 [13], for 3 740 of which there is a known specification violation, i.e., we expect a verifier to find a bug and document it with a violation witness, whereas no violation is known for the other 6 781, i.e., we expect a verifier to find a correctness proof and document it with a correctness witness.

We used CPAchecker and UAutomizer as verifiers for all of these tasks, but due to technical limitations not every validator supports all features required to analyze violation witnesses and correctness witnesses for each category of tasks. Table 6 depicts which task category is supported by which validator. [13] In SV-COMP 2019, no validator existed that supports the validation of correctness witnesses for the categories Concurrency and Termination.

6.3 Experimental Setup

Our experiments were conducted on machines with a 3.4 GHz 8-core CPU (Intel Xeon `E3-1230 v5`) with 33 GB of RAM. The operating system was Ubuntu `18.04` (64 bit), using Linux `4.15` and OpenJDK `1.8`. Each run for a single verification or validation task was limited to two CPU cores, a CPU run time of 15 min, and a memory usage of 15 GB. The benchmarks were executed using BenchExec [43] in version `1.17`.

6.3.1 Verifiers.

We used two verifiers, CPAchecker and UAutomizer. CPAchecker was used in version `cpachecker-1.7-witnesses-tosem-20181130` (revision 29 913 from the trunk). We configured it to use MathSAT5 as SMT solver. As in our preliminary work on correctness witnesses [23], we use k-induction with auxiliary-invariant generation for the tasks from category ReachSafety, as

[13]The benchmark definitions for all validators can be found at:
https://github.com/sosy-lab/sv-comp/tree/svcomp19/benchmark-defs.

Table 7. Confirmed and unconfirmed violation results in the category Concurrency

Validator	CPAchecker	
Producer	CPAchecker	Automizer
Confirmation rates:		
Produced	772	247
Confirmed	771	4
Unconfirmed	1	243
Confirmation rate	100 %	1.6 %

defined in configuration `svcomp18--kInduction`. For the other categories, which were not part of our preliminary evaluation, we use the corresponding analyses from the CPA-Seq submission for SVCOMP 2019, as defined in configuration `svcomp19--concurrency` for category Concurrency, `svcomp19--memorysafety` for category MemSafety, `svcomp19--overflow` for category Overflows, and `svcomp19--termination` for category Termination. UAutomizer was used in its SVCOMP 2019 version (`0.1.24-91b1670e`) with Z3 as SMT solver.

6.3.2 Validators. The categories supported by the validators is given in Table 6.

For the static validator based on CPAchecker, the same version as for the verifier was used with the configuration `witnessValidation`. This configuration performs violation-witness-based result validation by using CPAchecker's framework for configurable program analysis (cf. Sect. 4.1.1) to compose predicate analysis and explicit-state model checking into a combined analysis, as described in Sect. 4.2.2. To perform correctness-witness-based result validation, this configuration uses k-induction, where instead of synthesizing invariants itself like a verifier would, the validator uses only auxiliary invariants from the set of confirmed candidate invariants from the witness, as described in Sect. 4.3.2.

For the static validator based on UAutomizer, the same version as for the verifier was used and configured to perform witness-based result validation, which handles violation witnesses as described in Sect. 4.2.2 and correctness witnesses as described in Sect. 4.3.2.

CPA-witness2test was used in the same version of the CPAchecker framework as the CPAchecker-based verifier and was configured to perform the dynamic result validation described in Sect. 4.2.2.

FShell-witness2test was used in revision `c15c8acb` from its repository[14] and was configured to perform the dynamic result validation described in Sect. 4.2.2.

6.3.3 Presentation. All reported times (CPU time) are rounded to two significant digits. If the validation of a witness exceeds its resource limits before confirming the witness, then the validation result is counted as unconfirmed. The HTML tables in the reproduction package and on the supplementary web page (see Sect. 8) are generated with the table generator from BenchExec.

6.4 Results

6.4.1 Violation Witnesses. Table 6 shows that for violation witnesses, all four validators can be used, that the execution-based validators CPA-witness2test and FShell-witness2test support all categories except Concurrency and Termination, CPAchecker supports all categories, and UAutomizer supports all categories except Concurrency.

Claim 1: Consistency within the Same Framework. Our first experiment for violation witnesses represents a study showing that we were able to implement a witness exchange format for violation witnesses for C programs for CPAchecker and UAutomizer, where both can take the roles of

[14]https://github.com/tautschnig/fshell-w2t

Table 8. Confirmed and unconfirmed violation results in the category MemSafety

Validator	CPAchecker		CPA-witness2test		FShell-witness2test		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:								
Produced	107	67	107	67	107	67	107	67
Confirmed	106	54	17	15	29	1	27	43
Unconfirmed	1	13	90	52	78	66	80	24
Confirmation rate	99 %	81 %	16 %	22 %	27 %	1.5 %	25%	64%

Table 9. Confirmed and unconfirmed violation results in the category Overflows

Validator	CPAchecker		CPA-witness2test		FShell-witness2test		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:								
Produced	165	163	165	163	165	163	165	163
Confirmed	164	161	149	9	121	24	160	163
Unconfirmed	1	2	16	154	44	139	11	0
Confirmation rate	99%	99%	90%	5.5%	73%	15%	97%	100%

Table 10. Confirmed and unconfirmed violation results in the category ReachSafety

Validator	CPAchecker		CPA-witness2test		FShell-witness2test		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:								
Produced	964	491	964	491	964	491	964	491
Confirmed	920	271	698	218	554	184	634	438
Unconfirmed	44	220	266	273	410	307	330	53
Confirmation rate	95 %	55 %	72 %	44 %	57 %	37 %	66%	89%

Table 11. Confirmed and unconfirmed violation results in the category Termination

Validator	CPAchecker		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:				
Produced	575	558	575	558
Confirmed	568	432	557	548
Unconfirmed	7	126	707	10
Confirmation rate	99 %	77 %	97 %	98 %

a verifier (producing witnesses) and also, for categories where the corresponding tool supports validation, of a witness validator for their own witnesses. Additionally, because CPA-witness2test is also based on the CPAchecker framework, we expect CPA-witness2test to also be able to validate witnesses produced by the CPAchecker verifier.

Category Concurrency. The first column of Table 7 shows that in category Concurrency, CPAchecker confirmed 771 of 772 witnesses produced by CPAchecker, so that the confirmation rate for results produced by the same framework the validator is based on is almost 100 %.

Category MemSafety. The first, third, and last columns of Table 8 show that in category MemSafety, CPAchecker confirmed 106 of 107 witnesses produced by CPAchecker, that CPA-witness2test confirmed 17 of 107 witnesses produced by CPAchecker, and that Automizer confirmed 43 of 67 witnesses produced by Automizer, so that the confirmation rates for results produced by the same framework the validator is based on are 99 %, 16 %, and 64 %, respectively.

Category Overflows. The first, third, and last columns of Table 9 show that in category Overflows, CPAchecker confirmed 164 of 165 witnesses produced by CPAchecker, that CPA-witness2test confirmed 149 of 165 witnesses produced by CPAchecker, and that Automizer confirmed 163 of 163 witnesses produced by Automizer, so that the confirmation rates for results produced by the same framework the validator is based on are 99 %, 90 %, and 100 %, respectively.

Category ReachSafety. The first, third, and last columns of Table 10 show that in category ReachSafety, CPAchecker confirmed 920 of 964 witnesses produced by CPAchecker, that CPA-witness2test confirmed 698 of 964 witnesses produced by CPAchecker, and that UAutomizer confirmed 438 of 491 witnesses produced by CPAchecker, so that the confirmation rates for results produced by the same framework the validator is based on are 95 %, 72 %, and 89 %, respectively.

Category Termination. The first and last columns of Table 11 show that in category Termination, CPAchecker confirmed 568 of 575 witnesses produced by CPAchecker and that Automizer confirmed 548 of 558 witnesses produced by Automizer, so that the confirmation rates for results produced by the same framework the validator is based on are 99 % and 98 %, respectively.

We see that overall, we often achieve high confirmation rates if we apply validators to verification results produced by verifiers that are based on the same frameworks, although there is still some room for improvement regarding the validation of violation results by Automizer in category MemSafety (64 %). We attribute the lowest and third-lowest confirmation rates in this experiment, namely the 16 % achieved by CPA-witness2test for validating the results of CPAchecker in category MemSafety and the 66 % achieved by CPA-witness2test for validating the results of CPAchecker in category ReachSafety, to the fact that execution-based validators in general require very precise witnesses with concrete variable assignments for all input variables and otherwise fail, whereas model-checking-based validators, such as CPAchecker and Automizer, are often able to compute missing variable assignments during validation [26].

Claim 2: Validation across Frameworks. Our second experiment represents a study showing that we were able to communicate violation witnesses across frameworks, where verification results produced by the CPAchecker-based verifier are validated by the Automizer-based validator and vice versa, where verification results produced by the CPAchecker-based verifier and the Automizer-based verifier are validated by the dynamic validator FShell-witness2test, and where verification results produced by the Automizer-based verifier are validated by the dynamic validator CPA-witness2test.

Category Concurrency. The last column of Table 7 shows that this claim does not hold in category Concurrency: We see that CPAchecker confirmed only 1.6 % of the verification results produced by Automizer. We attribute this to the fact that Automizer only recently added support for verifying tasks in this category and has not yet fully implemented all features required to produce witnesses that can easily be validated. While this shows that there still remains work to be done to better support this combination, we chose to include the results for this part of the experiment for completeness and to accurately report the state of the art regarding available implementations.

Category MemSafety. Table 8 shows that in category MemSafety, while CPAchecker confirmed 81 % of the verification results produced by Automizer, Automizer confirmed only 25 % of the verification results produced by CPAchecker, which matches an observation from the previous experiment, namely that the support for the validation of violation results is still prototypical for the Automizer-based validator in category MemSafety. The results of the model-checking-based validators in the remaining categories are more promising, although it is expected that result validation across frameworks is more difficult than within the same framework. Table 8 also shows that CPA-witness2test confirmed 22 % of the verification results produced by Automizer, that FShell-witness2test confirmed 27 % of the verification results produced by CPAchecker, and that FShell-witness2test confirmed 1.5 % of the verification results produced by Automizer.

Category Overflows. Table 9 shows that in category Overflows, CPAchecker confirmed 99 % of the verification results produced by Automizer, that CPA-witness2test confirmed 5.5 % of the verification results produced by Automizer, that FShell-witness2test confirmed 73 % of the verification results produced by CPAchecker, that FShell-witness2test confirmed 15 % of the verification results produced by Automizer, and that Automizer confirmed 97 % of the verification results produced by CPAchecker.

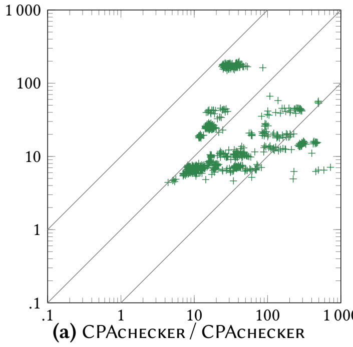
Category ReachSafety. Table 10 shows that in category ReachSafety, CPAchecker confirmed 55 % of the verification results produced by Automizer, that CPA-witness2test confirmed 44 % of the verification results produced by Automizer, that FShell-witness2test confirmed 57 % of the verification results produced by CPAchecker, that FShell-witness2test confirmed 37 % of the verification results produced by Automizer, and that Automizer confirmed 66 % of the verification results produced by CPAchecker.

Category Termination. Table 11 shows that in category Termination, CPAchecker confirmed 77 % of the verification results produced by Automizer and that Automizer confirmed 97 % of the verification results produced by CPAchecker.

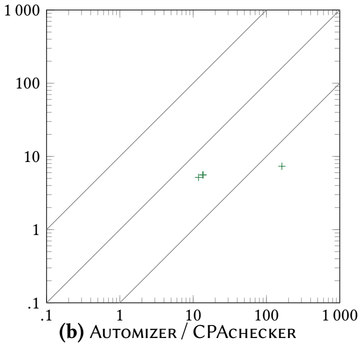
As observed in the previous experiment, the confirmation rates achieved by the execution-based validators CPA-witness2test and FShell-witness2test are mostly lower than those achieved by the model-checking-based validators CPAchecker and Automizer due to their requirement for more precise witnesses. For example, FShell-witness2test is only able to validate 1 of 67 results produced by Automizer in category MemSafety, CPA-witness2test is only able to validate 9 of 163 results produced by Automizer in category Overflows, and the confirmation rate of 37 % for FShell-witness2test validating the results of Automizer in category ReachSafety appears low if compared directly with the results achieved by the model-checking-based validators. On the other hand, FShell-witness2test is able to validate more results produced by CPAchecker than Automizer is able to validate in category MemSafety. Moreover, while there are generally fewer confirmations by the execution-based validators, these confirmations can be considered more valuable than the confirmations by model-checking-based validators in that they instill a higher confidence in the result and are bundled with easily debuggable executables for the verification result. For example, these numbers still show that the completely independent validator FShell-witness2test was able to synthesize executable binaries from 184 out of 491 witnesses produced by Automizer in category ReachSafety, execute them, and successfully replay the reported bugs, which gives a potential user not only a high confidence that these 184 bugs actually exist, but also provides observable, executable proofs for each confirmation.

Claim 3: Effectiveness and Efficiency of Validation Depends on Witness Contents. Our experiments also confirm that often, witness-based violation-result validation is faster than the corresponding preceding verification, although there are exceptions to this observation.

Category Concurrency. For example, Fig. 14a shows that in category Concurrency, using CPAchecker to validate verification results produced by CPAchecker is in most cases faster than the verification, but that there is also a small cluster of verification results where validation is almost ten times slower than the verification. For completeness, we also depict in Fig. 14b the scatter plot that compares the verification times of Automizer to the validation times of CPAchecker for Automizer's



(a) CPAchecker / CPAchecker



(b) Automizer / CPAchecker

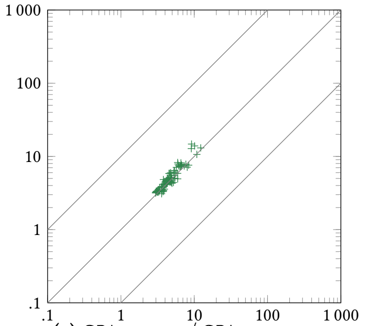
Fig. 14. Category Concurrency (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "*p*/*c*" abbreviates "witnesses produced by *p* that are confirmed by *c*"

results. However, even though in this figure, validation is always faster than verification, we do not consider the low number of validated results significant enough to draw any general conclusions.

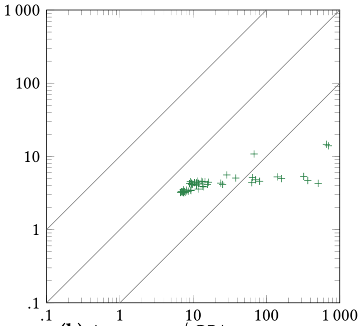
Category MemSafety. Figure 15a shows no significant time differences for category MemSafety between verifying a task with CPAchecker and validating the corresponding result with CPAchecker. On the other hand, Fig. 15b shows that verification results produced by Automizer are always validated faster by CPAchecker than they were produced. The same, however, is not true for the inverse case. Figure 15g shows that in fact, verification results produced by CPAchecker are always validated slower by Automizer than they were produced, and Fig. 15h shows only a few cases where validating verification results produced by Automizer are validated quicker by Automizer itself than they were produced. This matches our previous observation that the support for validating violation results is still prototypical for the Automizer-based validator. Figures 15c, 15d, 15e, and 15f, which are scatter plots for the verification results produced by CPAchecker and validated by CPA-witness2test, verification results produced by Automizer and validated by CPA-witness2test, verification results produced by CPAchecker and validated by FShell-witness2test, and verification results produced by Automizer and validated by FShell-witness2test, respectively, show that execution-based validation of results is mostly faster than verification although it must be noted that due to the low number of validations, this observation is not significant.

Category Overflows. Figure 16 shows for category Overflows almost no significant time differences for the model-checking-based validators between verifying a task and validating the corresponding verification result, which can be attributed to the fact that almost all tasks can already be verified in less than 10 s, i.e., very quickly, so that there is not much time to be gained by using a witness to reduce the search space during the validation. As for the category MemSafety, we again observe that the execution-based validators, are at least as quick and, in the case of FShell-witness2test, often even significantly faster than the corresponding verifications. However, the low number of confirmations again prohibits deriving a general claim from this observation.

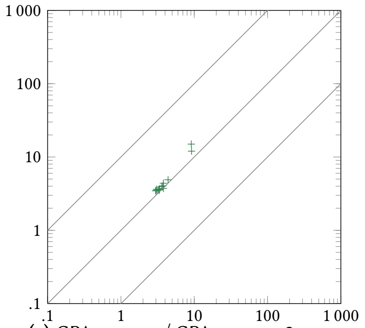
Category ReachSafety. Figure 17 shows a somewhat clearer picture for category ReachSafety: In Fig. 17a we can see that except for an insignificant amount of outliers, validating a verification result produced by CPAchecker with CPAchecker is at least as fast as producing that verification result, and that this effect appears to scale well, because even for many tasks where the verification took more than 100 s, validation took only less than a tenth of that time. In Fig. 17b we observe the same effect for verification results produced by Automizer and validated by CPAchecker, although not as pronounced as in the previous figure. Figures 17c, 17d, 17e, and 17f, which depict the verification



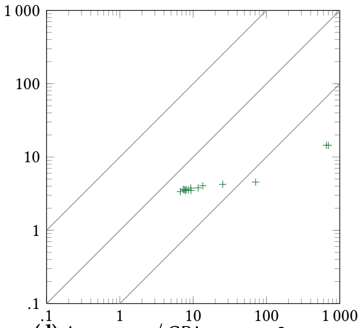
(a) CPAchecker / CPAchecker



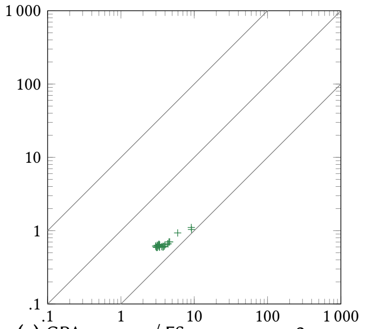
(b) Automizer / CPAchecker



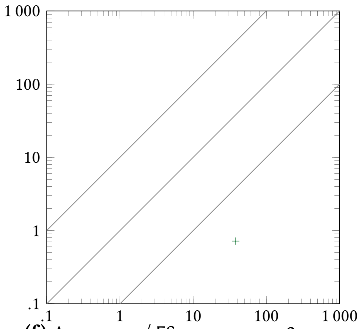
(c) CPAchecker / CPA-witness2test



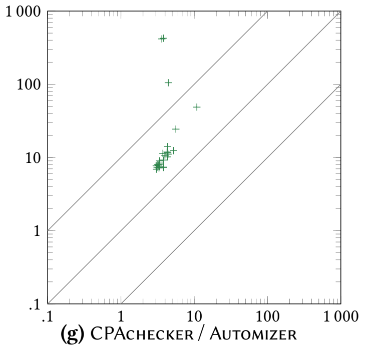
(d) Automizer / CPA-witness2test



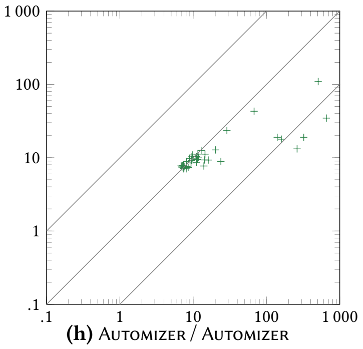
(e) CPAchecker / FShell-witness2test



(f) Automizer / FShell-witness2test

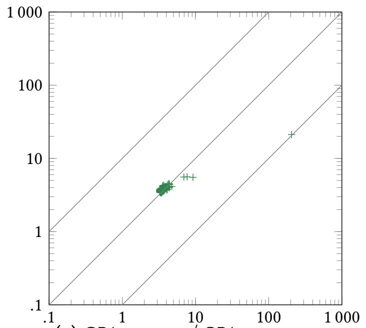


(g) CPAchecker / Automizer

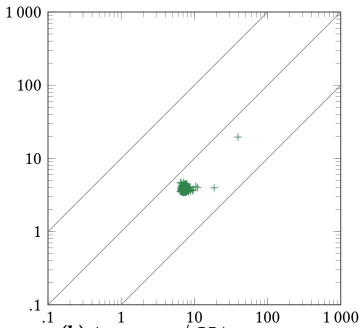


(h) Automizer / Automizer

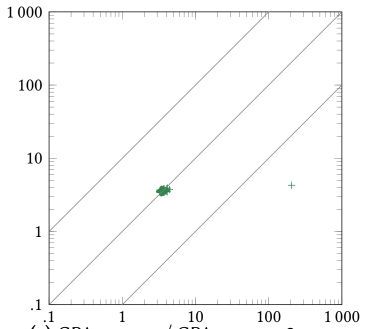
Fig. 15. Category MemSafety (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "*p*/*c*" abbreviates "witnesses produced by *p* that are confirmed by *c*"



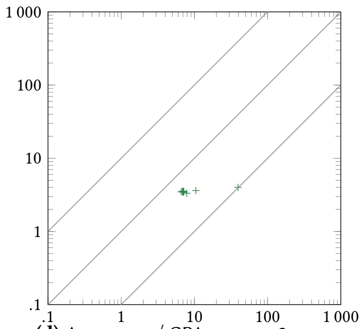
(a) CPAchecker / CPAchecker



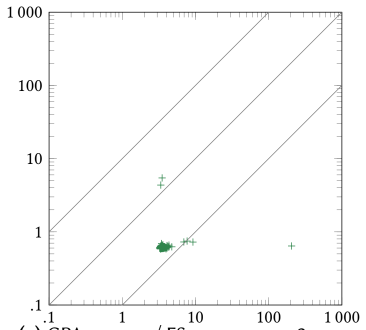
(b) Automizer / CPAchecker



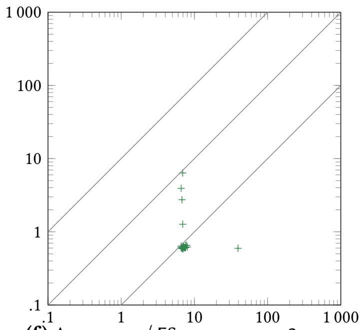
(c) CPAchecker / CPA-witness2test



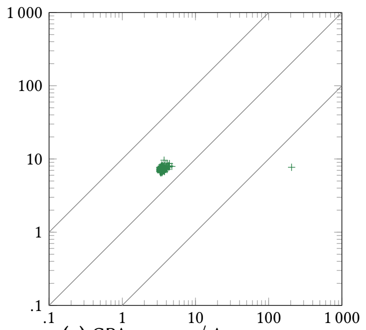
(d) Automizer / CPA-witness2test



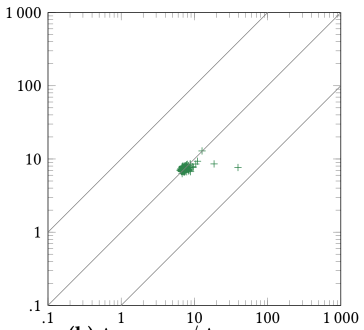
(e) CPAchecker / FShell-witness2test



(f) Automizer / FShell-witness2test



(g) CPAchecker / Automizer



(h) Automizer / Automizer

Fig. 16. Category Overflows (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "*p*/*c*" abbreviates "witnesses produced by *p* that are confirmed by *c*"

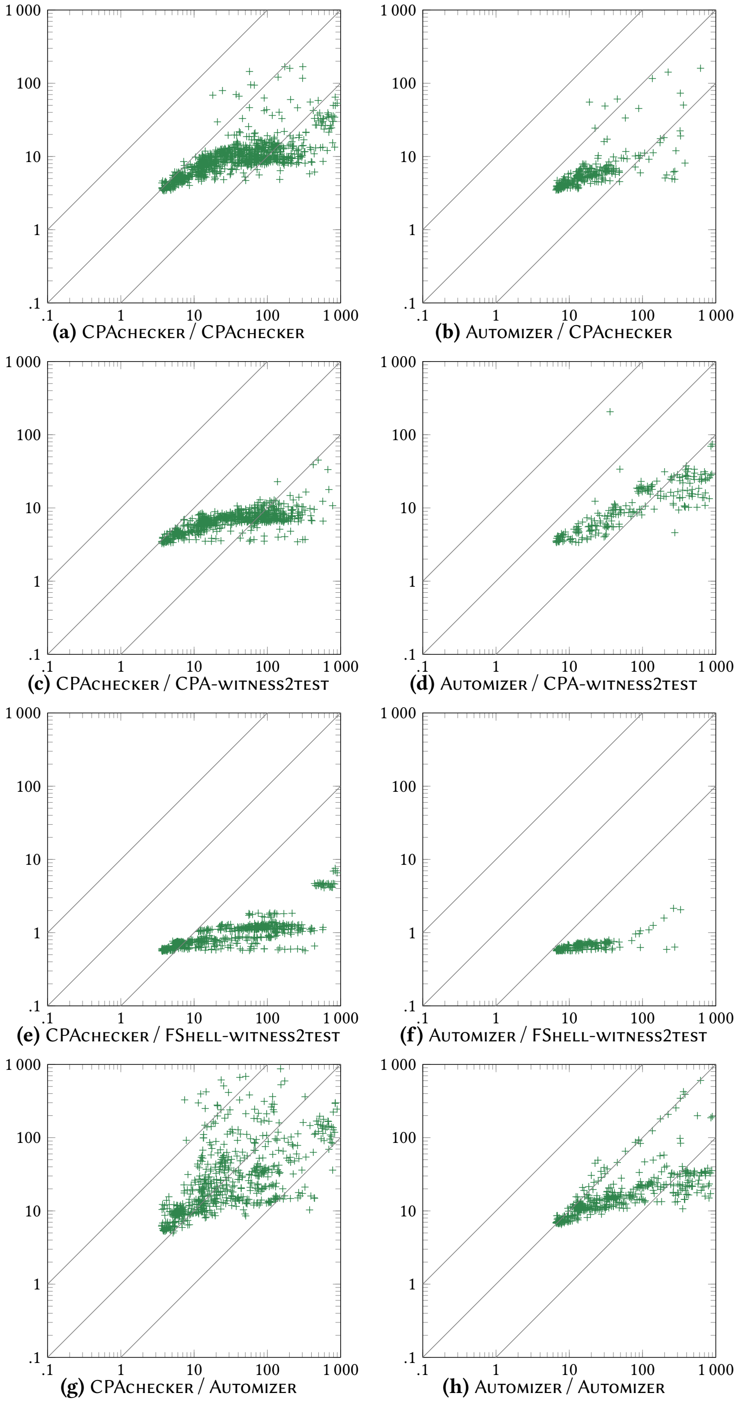


Fig. 17. Category ReachSafety (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c"

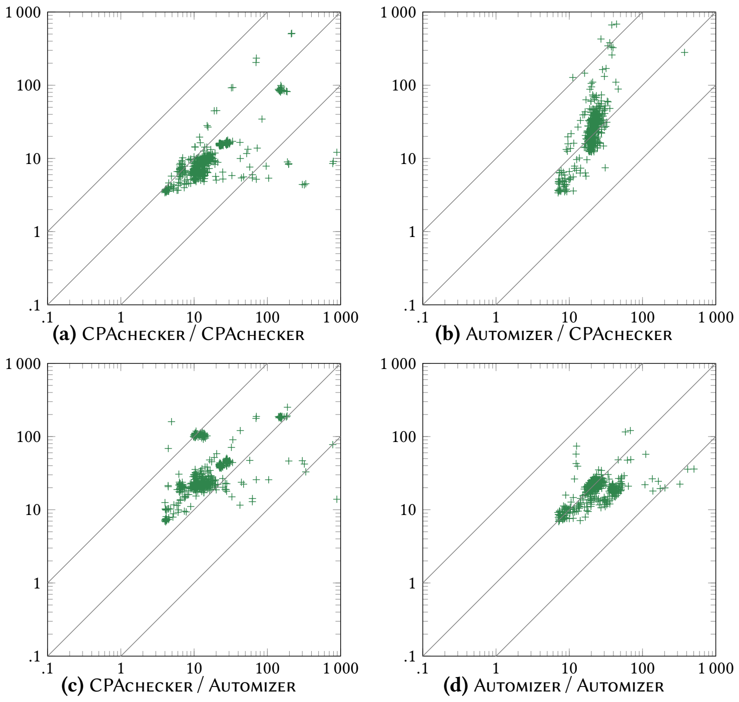


Fig. 18. Category Termination (violation): Scatter plots for pairwise composition for witness-based violation-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "*p*/*c*" abbreviates "witnesses produced by *p* that are confirmed by *c*"

results produced by CPAchecker and validated by CPA-witness2test, verification results produced by Automizer and validated by CPA-witness2test, verification results produced by CPAchecker and validated by FShell-witness2test, and verification results produced by Automizer and validated by FShell-witness2test, respectively, show that execution-based validation is usually significantly faster than verification, and often also faster than model-checking-based validation, even though fewer results can successfully be validated, which is particularly visible in Figs. 17e and 17f, which compare the validation times of FShell-witness2test to the corresponding verification times. Figure 17g, on the other hand, shows that applying the Automizer-based validator to the verification results produced by CPAchecker, there are cases where validation is slower than, as fast as, or faster than verification, with no clearly discernible trend, which means that Automizer apparently often does not profit from the reduced search space provided by the witnesses of CPAchecker and its validation times are more dependent on its own engine than on the witnesses, whereas Fig. 17h shows that Automizer can profit from its own witnesses, because the Automizer-based validator often validates a verification result in less time than Automizer took to produce it.

Category Termination. Lastly, for the category Termination, Figs. 18a and 18d show that both the CPAchecker-based validator and the Automizer-based validator profit from witnesses for verification results produced by their own respective frameworks and often validate these results in less time than it took to produce them, whereas there is no apparent performance improvement visible in Figs. 18b and 18c, which compare the validation times of CPAchecker for the results produced by Automizer and the validation times of Automizer for the results produced by CPAchecker, respectively, to the corresponding verification times.

Table 12. Confirmed and unconfirmed correctness results in the category MemSafety

Validator	Automizer	
Producer	CPAchecker	Automizer
Confirmation rates:		
Produced	118	108
Confirmed	52	108
Unconfirmed	66	0
Confirmation rate	44 %	100 %

Table 13. Confirmed and unconfirmed correctness results in the category Overflows

Validator	Automizer	
Producer	CPAchecker	Automizer
Confirmation rates:		
Produced	130	144
Confirmed	119	144
Unconfirmed	11	0
Confirmation rate	92 %	100 %

Summary. We observed that validation can be significantly faster than the preceding verification, but this effect is generally not guaranteed. In category ReachSafety, which is the largest of all five examined categories, we observe this effect even across verification frameworks. While these results are already promising, we interpret them as an indicator that more effort should be spent on improving witness-based validation of violation results, especially in the categories Concurrency, MemSafety, Overflows, and Termination, to achieve similar performance benefits as in category ReachSafety.

6.4.2 Correctness Witnesses. Table 6 shows that for correctness witnesses, only CPAchecker and UAutomizer can be used as validators, and that UAutomizer supports categories MemSafety, Overflows, and ReachSafety, whereas CPAchecker supports only category ReachSafety.

Claim 1: Consistency within the Same Framework. Our first experiment for correctness witnesses represents a study showing that we were able to implement a witness exchange format for correctness witnesses for C programs for CPAchecker and UAutomizer, where both can take the roles of a verifier (producing witnesses) and also, if supported, a witness validator for their own witnesses. The last columns of Tables 12 and 13 show that Automizer confirmed 108 of 108 witnesses produced by Automizer in category MemSafety and 144 of 144 witnesses produced by Automizer in category Overflows, so that the confirmation rates of its own witnesses are 100 % in both cases. The first and last columns of Table 14 show that CPAchecker confirmed 2 130 of 2 642 witnesses produced by CPAchecker, and that Automizer confirmed 2 694 of 2 749 witnesses produced by Automizer, so that the confirmation rates for their own witnesses are 81 % and 98 %, respectively. Furthermore, for the rejected witnesses, UAutomizer detects incorrect invariants in 14 of its own witnesses, and CPAchecker refutes none of its own witnesses [15]. ***Claim 2: Validation across Frameworks.*** Our second experiment represents a study showing that we were able to communicate correctness witnesses across

[15]It may be interesting to developers of other verifiers to learn that when the development of the CPAchecker-based correctness-witness export and validation started, there were a lot more incorrect invariants, which were caused by several actual bugs in other components of the framework that the CPAchecker team had been unaware of. In addition to the other benefits, implementing correctness-witness validation can therefore also be a way to improve the overall quality of a verifier.

Table 14. Confirmed and unconfirmed correctness results in the category ReachSafety

Validator	CPAchecker		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer
Confirmation rates:				
Produced	2 642	2 749	2 642	2 749
Confirmed	2 130	1 297	1 827	2 694
Unconfirmed	512	1 452	815	55
Confirmation rate	81 %	47 %	69 %	98 %

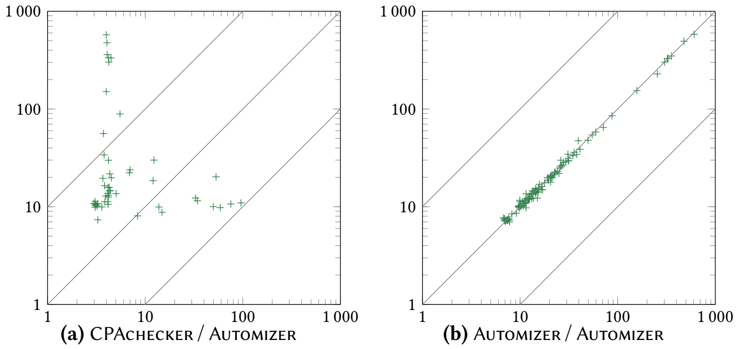


Fig. 19. Category MemSafety (correctness): Scatter plots for pairwise composition for witness-based correctness-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c"

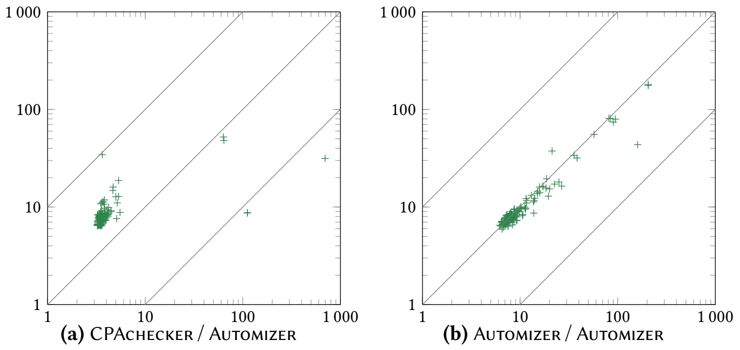


Fig. 20. Category Overflows (correctness): Scatter plots for pairwise composition for witness-based correctness-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "p/c" abbreviates "witnesses produced by p that are confirmed by c"

frameworks, where verification results produced by the CPAchecker-based verifier are validated by the Automizer-based validator and vice versa. Tables 12 and 13 show that Automizer confirmed 44 % of the verification results produced by CPAchecker in category MemSafety and 92 % of the verification results produced by CPAchecker in category Overflows. Table 14 shows that in category ReachSafety, CPAchecker confirmed 47 % of the verification results produced by Automizer, and that Automizer confirmed 69 % of the verification results produced by CPAchecker. Except for category Overflows, these numbers are not yet as favorable as those where the tools validate their own witnesses. We analyzed the unconfirmed results and found different causes for both cases: (1) CPAchecker did not

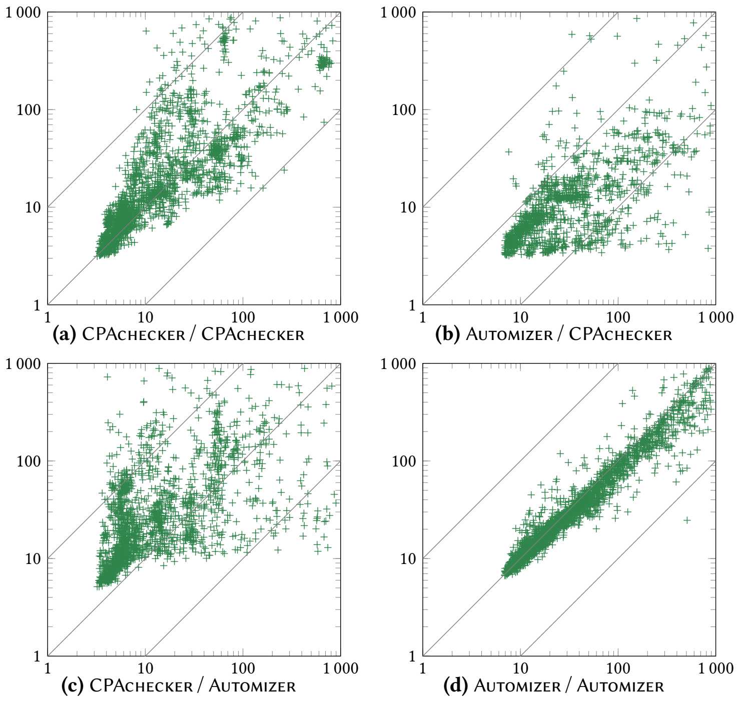


Fig. 21. Category ReachSafety (correctness): Scatter plots for pairwise composition for witness-based correctness-result validation; CPU seconds for producing a witness on the x axis, CPU seconds for result validation on the y axis; a caption "*p*/*c*" abbreviates "witnesses produced by *p* that are confirmed by *c*"

detect any incorrect invariants in the witnesses produced by Automizer, and there are often too few invariants present in those witnesses for the *k*-induction-algorithm to succeed within the time limit. This means that CPAchecker mostly does not dispute the witnesses of Automizer, but it cannot confirm them either. (2) Automizer is not always able to find the correct program location for an invariant. If Automizer maps an invariant to the wrong program location, and thus, the invariant does not hold there, then the witness is rejected. While there is still room for improvement to our implementations, in general, the witnesses were understood by the validators of other frameworks, and the rejections are mostly due to timeouts rather than due to wrong or miscommunicated invariants. Our experiment over the three categories MemSafety, Overflows, and ReachSafety, shows that for between 1 300 to 2 000 of 2 700 to 2 900 tasks verified by one verifier, a validator based on a different framework and different techniques not only agreed on the verdict but confirmed that no flaw was detected in the reasoning represented by the correctness witness, whereas previously, communicating such information between different tools was entirely impossible.

Claim 3: Effectiveness and Efficiency of Validation Depends on Witness Contents. Our experiments also confirm that the contents of the witnesses influences the difficulty of the validation, so that for a given verification task, one witness can lead to a quick validation, while a validation based on a different witness may require more resources or even fail to terminate at all. We first take a closer look at the differences in resource usage between verification and validation for a given task. Figure 21a shows that, especially for tasks that require more than 20 s of CPU time, CPAchecker produces three groups of witnesses, for which the validation is (a) about as fast as, (b) quicker than, and (c) slower than the preceding verification: The first group is explained by tasks for which few or even no auxiliary invariants are required by the k-induction technique. The second group is caused by tasks for which the witnesses contain useful invariants that allow the validator to quickly validate the task, while the verifier had to spend time on synthesizing the invariants. The third group represents tasks for which the witnesses contain significant amounts of invariants that turn out to be irrelevant, but the time spent by the validator to check them exceeds the time spent by the verifier to generate them. Figure 21b shows that many of the witnesses produced by Automizer that can be validated by CPAchecker are in most cases validated more quickly than they were produced. Figures 19a, 20a, and 21c are similar to Fig. 21a: there are cases for which the validation is faster than the verification and vice versa. But since in these three figures, validation and verification are performed by different tools, the differing characteristics of the two tools may outweigh the effects of the witnesses on validation speed: Automizer is often not faster at validating the invariants contained in the witnesses, and instead is often slower than CPAchecker for those of CPAchecker's witnesses that it can validate. It must also be noted that Fig. 20a shows that most of the few tasks in category Overflows that can be verified and where a validator confirms the result, are solved in less than about 10 s, which suggests that in this category, comparing verification and validation times is not particularly meaningful. Figures 19b, 20b, and 21d show that for Automizer, there is no discernible difference between the CPU times required to produce a witness and to validate it.

General Trend. In general, we could not observe a definite trend of speed-up over all validation runs using correctness witnesses. We attribute these results to the fact that it is not trivial to determine which invariants should be exported to the witness, because while exporting too much information unnecessarily complicates the validation, too few or too weak invariants may impede the feasibility of the validation. The fact that an invariant that suffices for one validator may not be sufficient for a different validator further complicates the decisions that drive the composition of invariants for a specific witness: suppose a verifier produces a witness that contains a 10-inductive invariant. A validator based on k-induction would likely be able to prove this invariant easily with $k = 10$, whereas a validator based on some other technique would likely have to first synthesize auxiliary invariants. We can, however, provide examples of cases for various different types of verification tasks for which a speed-up exists: [16] Table 15 shows for each supported combination of category, verifier, and validator an example for which the validation of a correctness witness was faster than the verification run that produced the correctness witness. For combinations where the validator is based on the same framework as the verifier (i.e., CPAchecker/CPAchecker, Automizer/Automizer), the speedup cannot be dismissed as caused by differences in the underlying implementation; instead, the speedup suggests that there is value in the guidance provided by the correctness witness in these cases. Unsurprisingly, validation only benefits from invariants that are difficult to derive but can be proved easily. If, however, too much work is left to the validator, then the validation is slower than the verification, because in addition to parsing the witness and matching its contents to the program, it also needs to synthesize its own invariants. Lastly, our implementations are based on generic model checkers and the potential for optimization towards validation is not yet utilized.

Summary. In conclusion, these experiments confirm that the contents of a correctness witness can be important for one of the validators (CPAchecker), while they do not seem to make noticeable difference for the other validator (Automizer), which can confirm more results but in turn is slower than the validator based on CPAchecker. This choice of a trade-off as to what constitutes an acceptable witness is one of the strengths of our flexible exchange format for correctness witnesses: Users may choose a quick but strict validator (rejects if invariant is too weak) or a slower but more tolerant one (constructs missing invariants), depending on their use case.

[16]We can pick the verification tasks from the bottom-right part of the scatter plots.

Table 15. Examples of verification tasks for which correctness-witness-based result validation was significantly faster than the verification run that produced the correctness witness

Program	Category	Verifier	Validator	Verifier CPU Time	Validator CPU Time
floppy_simpl4.cil.c	MemSafety	CPAchecker	Automizer	96 s	11 s
openbsd_cstrpbrk-alloca.i	MemSafety	Automizer	Automizer	38 s	34 s
Fibonacci02.c	Overflows	CPAchecker	Automizer	700 s	31 s
GopanReps-CAV2006-Fig1a.c.c	Overflows	Automizer	Automizer	160 s	44 s
minepump_spec2_product52.cil.c	ReachSafety	CPAchecker	CPAchecker	160 s	16 s
Problem15_label53.c	ReachSafety	CPAchecker	Automizer	720 s	98 s
test_locks_15.c	ReachSafety	Automizer	CPAchecker	300 s	6.6 s
tree.i	ReachSafety	Automizer	Automizer	510 s	25 s

6.5 Tutorial

In order to collect initial experience with the process of witness-based result validation, we list here a selection of tool invocations to get started with. The verification task that we use in this tutorial consists of the C program linear-inequality-inv-b.c from Fig. 9a and the specification unreach-call.prp for which the observer automaton is given in Fig. 2.[17]

Verify a Program with a Given Specification.
For CPAchecker, the following command line produces a witness similar to Fig. 22a:

```
scripts/cpa.sh \
  -spec sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

For UAutomizer, the following command line produces a witness similar to Fig. 22b:

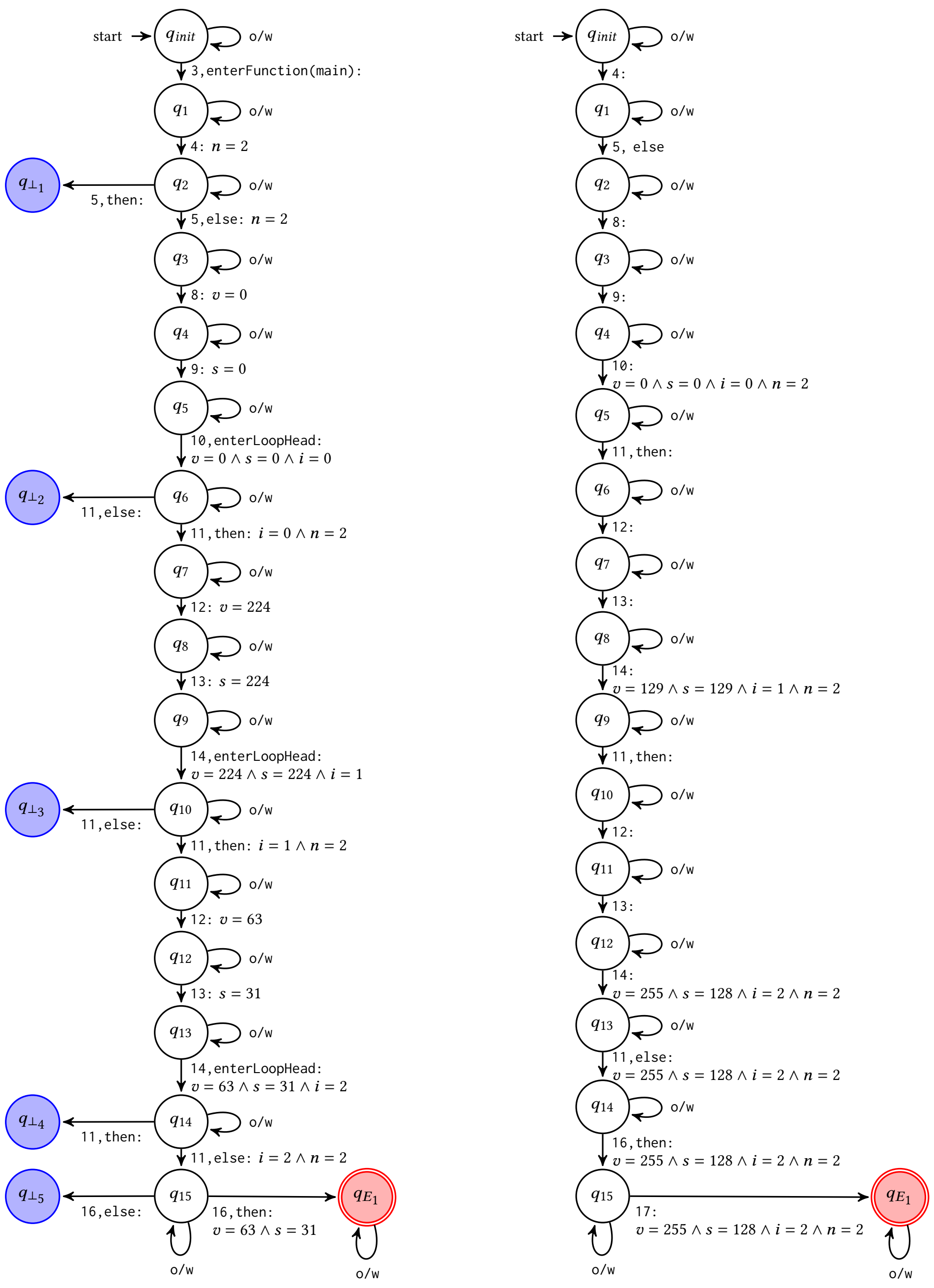
```
./Ultimate.py \
  --spec sv-benchmarks/c/properties/unreach-call.prp \
  --file sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c \
  --architecture 32bit
```

Validate the Result with the Produced Witness.
To attempt to validate a result using a witness `witness.graphml` with CPAchecker, execute the following command line:

```
scripts/cpa.sh \
  -witnessValidation \
  -witness witness.graphml \
  -spec sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

To attempt to validate a result using a witness `witness.graphml` with CPA-witness2test, execute the following command line:

[17]Note that between SV-COMP 2020 and SV-COMP 2021, the `unreach-call` specification changed from reachability of function `__VERIFIER_error` to function `reach_error` (see [18], page 404). If the goal is to use the task exactly as presented here, it is advisable to use the tool versions from the reproduction package and the specification file from the SV-COMP 2019 release of the benchmark repository. In general the latest versions of the tools can be used, as well as the latest version of the program and specification from the benchmark repository.


start
q_{init}
o/w
3,enterFunction(main):
q_1
4: $n = 2$
$q_{\perp_1}$
5,then:
q_2
5,else: $n = 2$
q_3
8: $v = 0$
q_4
9: $s = 0$
q_5
10,enterLoopHead: $v = 0 \wedge s = 0 \wedge i = 0$
$q_{\perp_2}$
11,else:
q_6
11,then: $i = 0 \wedge n = 2$
q_7
12: $v = 224$
q_8
13: $s = 224$
q_9
14,enterLoopHead: $v = 224 \wedge s = 224 \wedge i = 1$
$q_{\perp_3}$
11,else:
q_{10}
11,then: $i = 1 \wedge n = 2$
q_{11}
12: $v = 63$
q_{12}
13: $s = 31$
q_{13}
14,enterLoopHead: $v = 63 \wedge s = 31 \wedge i = 2$
$q_{\perp_4}$
11,then:
q_{14}
11,else: $i = 2 \wedge n = 2$
$q_{\perp_5}$
16,else:
q_{15}
16,then: $v = 63 \wedge s = 31$
q_{E_1}
start
q_{init}
4:
q_1
5, else
q_2
8:
q_3
9:
q_4
10: $v = 0 \wedge s = 0 \wedge i = 0 \wedge n = 2$
q_5
11,then:
q_6
12:
q_7
13:
q_8
14: $v = 129 \wedge s = 129 \wedge i = 1 \wedge n = 2$
q_9
11,then:
q_{10}
12:
q_{11}
13:
q_{12}
14: $v = 255 \wedge s = 128 \wedge i = 2 \wedge n = 2$
q_{13}
11,else: $v = 255 \wedge s = 128 \wedge i = 2 \wedge n = 2$
q_{14}
16,then: $v = 255 \wedge s = 128 \wedge i = 2 \wedge n = 2$
q_{15}
17: $v = 255 \wedge s = 128 \wedge i = 2 \wedge n = 2$
q_{E_1}
o/w

(a) Violation witness produced by CPAchecker **(b)** Violation witness produced by UAutomizer

Fig. 22. Violation witnesses produced by CPAchecker and UAutomizer for the verification task consisting of the C program linear-inequality-inv-b.c from Fig. 9a and the specification unreach-call.prp from Fig. 2

```
scripts/cpa_witness2test.py \
  -witness witness.graphml \
  -spec sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

To attempt to validate a result using a witness `witness.graphml` with FShell-witness2test, execute the following command line:

```
./test-gen.sh \
  --graphml-witness ../CPAchecker/witness.graphml \
  -m32 \
  --propertyfile sv-benchmarks/c/properties/unreach-call.prp \
  sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c
```

To attempt to validate a result using a witness `witness.graphml` with UAutomizer, execute the following command line:

```
./Ultimate.py \
  --spec sv-benchmarks/c/properties/unreach-call.prp \
  --file sv-benchmarks/c/loop-invariants/linear-inequality-inv-b.c \
  --validate ../CPAchecker/witness.graphml \
  --architecture 32bit
```

6.6 Validity

6.6.1 Benchmark Selection. For our benchmarking, we selected all full categories from the standard repository of software-verification tasks[18] without any restriction to subsets. Consequently, our experiments are performed over the largest openly available collection of verification tasks for the C programming language. For each category of the benchmark set, we show the results for all validators that support the category in 2019. While the main goal of this paper is to show that the approach can work in practice, we have not further excluded those verification tasks from the benchmark set for which the implementation is still insufficient, and instead show the results from the categories that are not yet well supported alongside those that are, to accurately represent the current state of the art in witness-based result validation, to pinpoint areas where further improvements are required, and to showcase the potential of witness-based result validation for areas where more mature implementations already exist.

Our knowledge about expected verification verdicts is based on the verdicts of the software-verification community.[18] In theory, it could be possible that an unconfirmed witness was not confirmed because the assumed bug does not exist, which is very unlikely because the benchmark sets is exposed to a lot of verification tools.

6.6.2 Verification Tools. Our implementations for producing and validating witnesses are based on several independent frameworks that use completely different technologies: CPAchecker implements a static approach to violation-witness-based result validation using a combination of predicate analysis and explicit-state model checking [39, 42], a static approach to correctness-witness-based result validation using k-induction [27], and a dynamic approach to violation-witness-based result validation (CPA-witness2test) that produces, runs, and checks executable tests [26]. UAutomizer uses an automata-based approach [84] to static witness-based result validation. FShell-witness2test is based on the test-vector format of FShell [89] and is independent from any model-checking framework. This means that while comparisons of speed between verification with one tool and validation with the other tool are only meaningful on a very coarse level, we can show that a wide variety of techniques can be used for witness-based result validation.

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

6.6.3 Reproducibility. All data presented, including verification tasks, witnesses, verifiers, and their configurations, are available on our supplementary web site (see Sect. 8). For controlling and measuring the computing resources used in our experiments, such as memory, CPU time, core and memory assignment, we use the state-of-the-art benchmarking framework BenchExec [43], and thereby ensure that our results are accurate, reliable, and reproducible. To further improve reproducibility of our experiments, we also selected the configurations of the verifiers that are used to produce verification results and witnesses with a focus on the stability of their results instead of on their general effectiveness. For example, for CPAchecker, a more effective configuration than the one used in our experiments was used in SV-COMP 2019 in category ReachSafety, but since this configuration uses timers to dynamically switch between various analyses, its results are less stable than our choice of a single analysis, k-induction, for this category.

7 RELATED WORK

The exchange format for verification witnesses described in this article and the corresponding techniques for communicating verification witnesses across verification tools were introduced —initially only for violation witnesses— in 2015 [25]. In 2016, the format was extended to encompass correctness witnesses [23]. We give updated technical descriptions and evaluation results for existing witness-based result validators [23, 25, 26]. [19] The flexibility, stability, and practical applicability of the exchange format are evidenced by the fact that it has already been successfully applied for several years now in the annual TACAS International Competition on Software Verification (SV-COMP) [10, 11, 12]. As a result, all competing verifiers now support the exchange format for verification witnesses and augment their verification results with it.

7.1 Exchange Formats

Before the common exchange format became available, verification witnesses were used only based on proprietary formats within particular tools. For example, Esbmc was extended to reproduce errors via instrumented code [117], and CPAchecker was used to validate previously computed error paths by interpreting them as witness automata that guide and restrict the state-space search [47]. There are other exchange formats as well: (1) The Certification Problem Format (CPF) [121] is used by the competition on termination [75] to store termination proofs for term-rewrite systems. (2) The DRAT [87] format is used in the SAT competitions [8] since several years in order to validate the correctness of proofs of unsatisfiability of a propositional formula using a witness validator for DRAT [128]. (3) The Static Analysis Results Interchange Format (SARIF) [115] is used to represent results from static analysis by some industrial tools, such as CodeSonar [20], SWAMP [21], and Visual Studio [22], and which is mainly intended as input for visualization tools and for aggregating and embedding analysis results into bug-tracking or continuous-integration systems rather than for semantic analysis such as result validation.

7.2 Certifying Algorithms

The concept of *certifying algorithms* [107] is a solution for increasing trust in the results produced by potentially complex and error-prone computations. The paradigm of certifying algorithms demands that each algorithm provides, together with the computed output, a witness that in turn can be used to verify that the output is indeed a correct solution for the given input problem.

[19] MetaVal [44] and NitWit [122] are not included in our evaluation because they were developed after our evaluation was done in 2019.
[20] https://www.grammatech.com/products/codesonar
[21] https://github.com/mirswamp/deployment (see also https://continuousassurance.org/mir-swamp)
[22] https://visualstudio.microsoft.com

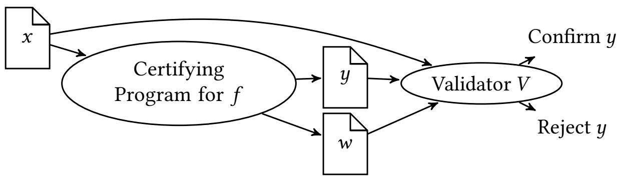


Fig. 23. A certifying program for a function f computes $y = f(x)$ and produces a witness w, which is then used to check the correctness of result y by a witness validator; adapted from [107]

Figure 23 illustrates this workflow for a certifying program for a function f, i.e., an implementation of a certifying algorithm: The program receives the input x, computes the result $y = f(x)$, and produces the witness w. The witness validator V receives the inputs x, y, and w, and leverages w to determine whether y is a correct result for $f(x)$. By certifying each individual result, the more difficult problem of proving the correctness of the certifying algorithm or its implementation is avoided. This concept applies to both violation witnesses and correctness witnesses: A violation witness is a certificate for a specification violation found by a verifier, whereas a correctness witness is a certificate for the proof found by a verifier. Both can be used by a validator to try to re-establish the verification result. The core advantages of using this approach are that to trust the verification result, it is not necessary to trust the producer of the witness, and that the result validator can re-establish the result independently. In fact, with verification witnesses, the tool that is used for the witness-based validation of a result can even work with a different abstract domain than the tool that produced the result and witness [111].

7.3 Counterexamples

When a verifier detects a bug, it usually provides some form of counterexample [7, 12, 19, 57, 60, 61, 103]. On top of that, however, there is a growing demand for quick and automatic validation of program error paths to raise the confidence in automatically detected bug reports [12, 47, 68, 111], most importantly to reduce the number of false alarms. For example, an expensive, high-precision feasibility check can be used to filter out false alarms produced by an efficient, low-precision data-flow analysis within one instance of a verifier [68]. Experiments show that instead of repeating a full verification task from scratch, it is usually significantly faster to validate an existing verification result using a violation witness [25, 47]. However, without a unified exchange format for violation witnesses to export counterexamples and use them as input to another tool, full programs were synthesized from counterexamples and used as a medium [33, 38, 117]. For witness-based result validation this approach is not useful, because the result and witness need to be checked against the original, unchanged program to ensure that no new error paths were introduced that did not exist originally. In the context of distributed high-performance computing, some exchange-formats for system traces exist, e.g., the MPI trace format [1], or the Open Trace Format [72, 102, 126], whose primary purpose is to keep record of system events, such as messages that are exchanged between processes. These formats strongly focus on distributed systems with time-stamped events and are not applicable to our problem. Many applications for violation witnesses already exist [3, 22, 62, 71, 78, 79, 81, 98, 104, 131], and a common format that can be used to exchange witnesses across verification tools will stimulate further research in this direction, particularly on combinations of verification, debugging, and visualization techniques.

7.4 Test-Case Generation

Verification counterexamples have been used to *generate test cases* for two decades now [19, 90, 91, 125]. Various automatic test-case generation techniques have been developed as extensions of this idea [80, 97, 119] and as combinations of counterexample-based test-case generation with other techniques, such as random testing [76, 106]. Test-goal automata are used to achieve specific coverage or to reach test goals by leading a program analysis towards specific program locations [35, 52]; conceptually, test-goal automata are simply a specific use case for the violation-witness automata that we present. Test cases from verification counterexamples have also been used to create debuggable executables [110, 117]. Two of the violation-witness-based result validators that we present [26] use this idea to validate verification results by synthesizing an executable from a verification task and a violation witness, and executing it to check if the reported error actually occurs. By using the common exchange format for witnesses, this technique can be applied to synthesize executables using the verification results of any tool that supports the common format. While other counterexample-based approaches for generating executable test cases [53, 67, 73, 105] are limited to concrete and tool-specific counterexamples, we do not require full counterexamples of any specific verifier; instead, our approach works on more flexible —and, thanks to our concept of witness refinement, potentially abstract— violation witnesses.

7.5 Correctness Certificates

There is a long history of *correctness certificates* for the purpose of increasing the trust in code that is generated from some form of formal description or model (e.g. [54, 58, 85, 94, 129]). While there are efforts to reduce the often inconveniently large size of these proofs [74], these correctness certificates are still complete proofs of functional correctness. While our exchange format can also be used as correctness certificate and to represent a full proof, this is not required: a correctness witness is more general, in that it can also be used as a *partial* proof of correctness [23], which can be more concise than a full proof. Alan Turing suggested already in 1949 to annotate programs with assertions "from which the correctness of the whole program easily follows." [124]

7.6 Proof-Carrying Code (PCC)

One application of correctness certificates has previously been explored in the context of *proof-carrying code* (PCC) [112]. PCC is a mechanism where an untrusted source supplies an executable program and a correctness certificate, both of which are therefore also untrusted initially. However, trust can be established by using a trusted validator to check the witness against the program and specification. Certifying model checkers can use the intermediate results of their verification procedure to compose full proofs and export them as proof certificates [111].

The exchange format for correctness witnesses allows the mechanism of proof-carrying code to be applied to real-world C programs and enables further verification tools to adopt the technique. Compared to previous publications on proof-carrying code, the main advantage that our exchange format and validation techniques provide is that we do not strictly require the witness to contain a full proof. We found that in practice, a complete proof for even short programs with simple specifications may become prohibitively large in size unless a considerable amount of additional effort is spent on simplifying formulas. Especially for more complex verification tasks, it is often neither desirable nor even feasible to handle such a full proof — as in mathematics, concise lemmas or proof sketches are priceless. [23] Consequently, we support flexibility: Given two witnesses w_1 and w_2, we consider w_1 to be of higher quality than w_2 if a witness-based result validator can more quickly re-establish the verification result using w_1 than using w_2. A less detailed witness may still succeed in guiding the validator to the proof, but in turn may require more effort from the validator. Another difference to classic PCC is that we consider the witness as its own, separate, first-class object, and do not use the program to carry the proof, thereby following the best practice of separation of concerns, which leads to higher flexibility and maintainability.

[23]The proof for the Schur-Number-Five problem is larger than 2 PB [88].

7.7 Reusing Reachability Graphs

The intermediate results produced by model checkers during their state-space exploration are often materialized as an *abstract reachability graph* (ARG) [31], which consists of the abstract states found by the model checker and the program transitions between those states. The ARG is the basic data structure in tools like Blast and CPAchecker, and can be used as a source of invariants of the program [85], which in turn can be used for PCC, or for extreme model checking [86]. Extreme model checking checks if a previously computed ARG is also still a safety proof for a given, slightly modified, input program. Slab [70] is a certifying model checker that produces a proof certificate for the abstract model of a program in SMT-LIB format. While such a certificate can easily be checked using an SMT solver, mapping it back to the original program [24] to validate that it really certifies the correctness *of the original program* is non-trivial. As a result, even if checking the SMT-LIB certificate with an SMT solver produces the expected answer, a user still has no way to confirm that the certificate faithfully refers to the original program.

7.8 Search-Carrying Code (SCC)

The concept of *search-carrying code* (SCC) [123] shares with verification witnesses the essential idea of reconstructing a verification result by guiding a validator through the state space of a program. For this purpose, SCC uses search scripts that guide a model checker along paths of the ARG. Search scripts can be seen as a special instance of the generic concept of correctness witnesses where all invariants are omitted and the validator uses only the branching information from the witnesses as a suggestion to guide its state-space exploration, potentially saving time by simply confirming the suggested ARG rather than having to spend effort determining it itself from scratch. In comparison to search scripts, witnesses overcome the following three limitations (cf. Sect. 4.3 in [123]): (i) While SCC is bound to explicit-state model checking, the verification-witness exchange format is independent from the verification approach. (ii) The search scripts used by the existing implementation of SCC depend on a very specific transition-statement interpretation of Java Pathfinder (JPF), whereas verification witnesses allow a flexible mapping from program operations to the verifier-specific states and transitions that is even tolerant to code reduction, i.e., gaps in the witness that correspond to program code on which the producing verifier did not provide any information. (iii) Due to the reasons above, SCC is only supported by JPF, whereas the exchange format for verification witnesses is designed to work across different verifiers, even if they rely on different technologies, as shown by the widespread adoption of the format [12]. For practical impact, we have found these extensions to be essential.

7.9 Proof Programs and Configurable Certification

An important aspect of PCC is the goal that validation should be significantly faster than verification. In programs-from-proofs [96], correctness certificates are materialized as new programs that are behaviorally equivalent to the corresponding input program and are generated by a predicate analysis. Although they may be exponentially larger in terms of lines of code, these new programs can be verified by using a less expressive and more efficient data-flow analysis. Certificates for configurable program analysis [95, 96] consist of all reachable states of a program, which is comparable to a correctness witness where the reachable states are encoded as invariants at each program location. Various size-reducing techniques are then applied to reduce space consumption and I/O, and to speed up the validation. Because correctness witnesses do not require full proofs but can also contain partial proofs, a validator may choose to apply its own verification strategy

[24]In real-world scenarios, the original program is usually not given as a formal transition system with a well-defined one-to-one variable mapping to SMT-LIB, but must first be transformed by the verifier.

to complement a partial proof or even perform the complete verification of the full verification task itself. As a consequence, correctness-witness-based validation of verification results does not necessarily exhibit a speedup. Nevertheless, in scenarios where the witnesses do represent complete proofs, similar techniques can be applied and speedups can be achieved. The size of correctness witnesses is generally not an issue in our case, because both implemented witness producers, CPAchecker and UAutomizer, restrict themselves to loop invariants and procedure post conditions instead of exporting invariants for every program location.

7.10 Partial Verification and Cooperative Verification

Software verifiers have three possible outcomes: they either (1) prove correctness, (2) detect a bug, or (3) fail. Correctness witnesses [23] and violation witnesses [25] address the first and second case, respectively. To complete the picture, *conditional model checking* (CMC) [32] addresses the third case. The idea of CMC is to provide reports of partial verification results in case full verification fails: An output condition describes the result of an incomplete verification attempt, i.e., which parts of the state space have already been verified successfully, and an input condition instructs a model checker to restrict the verification of a system, i.e., it describes which parts of the state space are left to be verified. To complete the verification, subsequent verification runs with a different approach can then use the output condition of the previous run as an input condition to simplify their task. Various concepts to represent the conditions, such as assumption automata [32] or execution reports [56], have been explored in existing implementations of CMC. Recently, the concept of reducer-based construction of conditional verifiers was introduced to facilitate the adoption of CMC [37]: A *reducer* synthesizes, from a given input program and input condition, a new *residual* program that consists of only those parts of the original input program that are still to be verified, according to the input condition. Any off-the-shelf verifier can then be used for the conditional model checking of the residual program. As an alternative, verification witnesses could be used as a medium for CMC, by describing (a) paths (in violation witnesses) that hindered a complete verification and (b) invariants (in correctness witnesses) that were used to verify the part of the system that was successfully verified.

In cooperative verification [45], these techniques are leveraged to solve verification tasks by sharing information between different verification approaches and tools, not necessarily unidirectionally, but potentially even back and forth between components, and over multiple iterations [36].

7.11 Generalization

Verification witnesses subsume several of the previously known types of verification artifacts. We try to explain this using Figure 24. Firstly, we consider the two main types of witnesses disjoint, that is, a verification witness is either a violation witness or a correctness witness. This design choice is not obvious, because it is arguable why a violation witness should not contain invariants that help rule out considering infeasible error paths during the validation. Secondly, both witness types allow for a range of abstraction levels. A violation witness can be as abstract as an abstract counterexample from model checking (Sect. 7.3) on the one hand (abstract extreme: no restriction of data values, example: Fig. 9b) and it can be as concrete as a test case (Sect. 7.4) on the other hand (concrete extreme: all data values concretely given, example: Fig. 9d). But violation witnesses can have any level of abstraction in between the two extreme cases (intermediate: intervals for data values, example: Fig. 9c). Similarly, a correctness witness can be as abstract as in search-carrying code (Sect. 7.8) (only guiding the validator through the state space) and as concrete as in proof-carrying code (Sect. 7.6) (providing all proof ingredients). There is a wide spectrum of possibillities in between, for example certificates (Sects. 7.5 and 7.9).

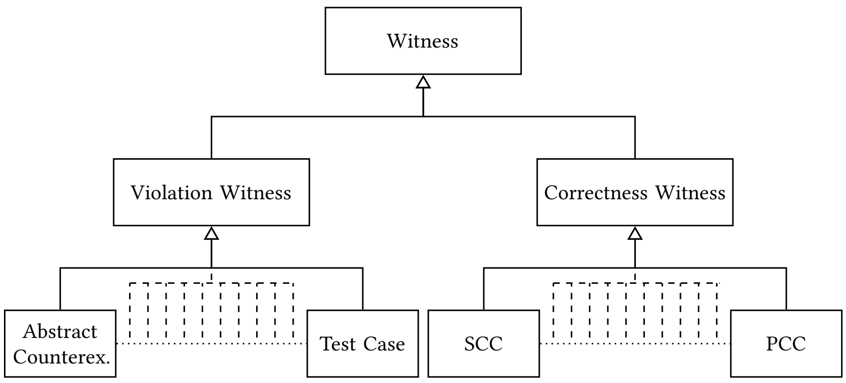


Fig. 24. Classification of different types of witnesses

8 CONCLUSION

Software verification in general is an undecidable problem. Therefore, effectiveness and efficiency have always been two main concerns of software verification, i.e., the goal was to make software verification solve more problems, and solve them quicker, and hence, there have been many breakthroughs in the past decades that made software verification efficient enough to be applicable on an industrial scale. However, even though effectiveness and efficiency are certainly valid and important concerns, an oft-repeated argument against practical application of software verification is fear of the significant economic disadvantages caused by time wasted on the investigation of false alarms and of the potentially catastrophic consequences of misplaced trust in proofs that may turn out to be wrong. This fear is reinforced by a lack of usability of the results: A simple TRUE-or-FALSE answer to a verification problem is insufficient to understand and validate the result.

On the other hand, testing is an established method for software-quality assurance, because its results are concrete and tractable: An engineer constructs a test suite for a given coverage goal, executes the tests, and obtains precise and graspable results: (i) a quantitative coverage and (ii) a qualitative answer to the question which tests passed and which tests failed. This process requires considerable resources to be spent, but in return, concrete answers are provided, and, contrary to classic software verification, the interpretation of these results does not require an academic education.

If we compare the testing process to verification, we must acknowledge that in classic verification, an engineer also has to invest a significant amount of resources, as in testing, but in turn gets back only an oversimplified answer TRUE or FALSE without any argument or explanation. The confidence in this answer is usually only derived from the reputation of the verification tool, because manually inspecting an error path for the verification answer FALSE to determine whether it represents an actual bug or a false alarm is a tedious task and a waste of expensive developer time. To make matters worse, most classic tools did not even bother to give an explanation why the verifier reports the program as correct when its answer to the verification problem is TRUE.

We aim to change this situation and propose using tool-independent and machine-readable witnesses as a richer, more valuable form of verification result for both specification violations and correctness. In this paper we presented a formalism to express both violation witnesses and correctness witnesses, while also outlining their necessary differences. We suggest a concrete format to represent such witnesses for verification results for tasks derived from C programs and present four different implementations of validators that support this format. We believe that producing witnesses should be easy, because in order to find a bug, a useful verifier should already be able to give the user a test case or a concrete error path, and any verifier designed for more than just falsification, i.e., hunting bugs, must also already derive some form of a proof of correctness. In practice, of course, there certainly are some engineering efforts required to construct a useful witness. Witness-based result validation, on the other hand, is more difficult to implement than witness construction: the validator must not only understand the assumptions and invariants in the witness, but also correctly assign them to the program states that they were intended for. The formalism presented in this paper shows one possible approach for achieving this task, but if its direct implementation in a given verification framework is infeasible, the approach can be adapted, as exemplified by the different implementations that we showed.

We performed an extensive experimental study with thousands of verification runs on tasks from the largest public repository of verification problems (C programs). We implemented our validation approach in four result validators that have already been used for this purpose in the recent competitions on software verification, and have applied these validators to results produced by two verification tools that have achieved top scores in these competitions for years. The results obtained by our proof-of-concept implementations demonstrate that the proposed approach can work in practice. Since the advent of witnesses a few years ago, others have implemented support for witnesses in their tools. We hope that this process continues and that more developers find our ideas useful, thus adding the value of diversity to the concept: Although it may serve as a sanity check to apply a validator based on a certain framework to a witness produced by a verifier built on those same components, flaws in the reasoning may inadvertently be covered up by a common defective component. Our solution is to instead establish a common exchange format supported by many verifiers, such that different result validators based on different technologies can be leveraged. In the meanwhile, there are eight published validators for C programs (CPAchecker [25], CPA-witness2test [26], Dartagnan [116], FShell-witness2test [26], MetaVal [44], NitWit [122], Symbiotic-Witch [4], and UAutomizer [25]), which are based on seven completely different technologies, and our current results on witness validation demonstrate that diversity is beneficial. In SV-COMP 2022, two validators for Java progams (GWit [92] and Wit4Java [130]) were introduced. Establishing witnesses as an accepted standard in software verification will serve to open tools up to other uses besides plain verification and validation, such as quality measures for invariants or error paths, witness visualization, witness maintenance, databases for bugs and proofs, regression verification, and many more.

DECLARATIONS

Data-Availability Statement. All results, tools, and verification tasks that we used in our evaluation are available on the supplementary web page[25] and in a reproduction package at Zenodo [24]. In addition to the experimental results that we produced for this article, there are publicly available results for all verifiers and validators that participated in the competition on software verification (SV-COMP). A complete set of violation and correctness witnesses from the verifiers that participated in SV-COMP is available for 2019 [15] and for 2020 [17]. A statistical overview of the witnesses produced in SV-COMP 2019 and a description of the witness-store format that is used for those archives is also available [14]. Some of the results are shown in aggregated form in the competition reports, in Table 10 of the 2020 report [16], in Table 10 of the 2019 report [13], in Table 8 of the 2017 report [12], and in Table 7 of the 2016 report [11].

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

Acknowledgements. We thank Martin Spiessl and the anonymous reviewers for their careful proof-reading and suggestions for improvement.

[25]https://www.sosy-lab.org/research/verification-witnesses-tosem/

REFERENCES

[1] L. Alawneh and A. Hamou-Lhadj. 2011. MTF: A Scalable Exchange Format for Traces of High Performance Computing Systems. In *Proc. ICPC*. IEEE, 181–184. https://doi.org/10.1109/ICPC.2011.15

[2] J. Alglave, A. F. Donaldson, D. Kröning, and M. Tautschnig. 2011. Making Software Verification Tools Really Work. In *Proc. ATVA (LNCS 6996)*. Springer, 28–42. https://doi.org/10.1007/978-3-642-24372-1_3

[3] C. Artho, K. Havelund, and S. Honiden. 2007. Visualization of Concurrent Program Executions. In *Proc. COMPSAC*. IEEE, 541–546. https://doi.org/10.1109/COMPSAC.2007.236

[4] P. Ayaziová, M. Chalupa, and J. Strejček. 2022. Symbiotic-Witch: A Klee-Based Violation Witness Checker (Competition Contribution). In *Proc. TACAS (2) (LNCS 13244)*. Springer.

[5] T. Ball, V. Levin, and S. K. Rajamani. 2011. A Decade of Software Model Checking with Slam. *Commun. ACM* 54, 7 (2011), 68–76. https://doi.org/10.1145/1965724.1965743

[6] T. Ball and S. K. Rajamani. 2002. *SLIC: A Specification Language for Interface Checking (of C)*. Technical Report MSR-TR-2001-21. Microsoft Research. https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/

[7] T. Ball and S. K. Rajamani. 2002. The Slam project: Debugging System Software via Static Analysis. In *Proc. POPL*. ACM, 1–3. https://doi.org/10.1145/503272.503274

[8] T. Balyo, M. J. H. Heule, and M. Järvisalo. 2017. SAT Competition 2016: Recent Developments. In *Proc. AAAI*. AAAI Press, 5061–5063. https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14977

[9] D. Beyer. 2012. Competition on Software Verification (SV-COMP). In *Proc. TACAS (LNCS 7214)*. Springer, 504–524. https://doi.org/10.1007/978-3-642-28756-5_38

[10] D. Beyer. 2015. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In *Proc. TACAS (LNCS 9035)*. Springer, 401–416. https://doi.org/10.1007/978-3-662-46681-0_31

[11] D. Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *Proc. TACAS (LNCS 9636)*. Springer, 887–904. https://doi.org/10.1007/978-3-662-49674-9_55

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

[13] D. Beyer. 2019. Automatic Verification of C and Java Programs: SV-COMP 2019. In *Proc. TACAS (3) (LNCS 11429)*. Springer, 133–155. https://doi.org/10.1007/978-3-030-17502-3_9

[14] D. Beyer. 2019. A Data Set of Program Invariants and Error Paths. In *Proc. MSR*. IEEE, 111–115. https://doi.org/10.1109/MSR.2019.00026

[15] D. Beyer. 2019. Verification Witnesses from SV-COMP 2019 Verification Tools. Zenodo. https://doi.org/10.5281/zenodo.2559175

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

[17] D. Beyer. 2020. Verification Witnesses from SV-COMP 2020 Verification Tools. Zenodo. https://doi.org/10.5281/zenodo.3630188

[18] D. Beyer. 2021. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Proc. TACAS (2) (LNCS 12652)*. Springer, 401–422. https://doi.org/10.1007/978-3-030-72013-1_24

[19] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating Tests from Counterexamples. In *Proc. ICSE*. IEEE, 326–335. https://doi.org/10.1109/ICSE.2004.1317455

[20] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. The Blast Query Language for Software Verification. In *Proc. SAS (LNCS 3148)*. Springer, 2–18. https://doi.org/10.1007/978-3-540-27864-1_2

[21] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. 2009. Software Model Checking via Large-Block Encoding. In *Proc. FMCAD*. IEEE, 25–32. https://doi.org/10.1109/FMCAD.2009.5351147

[22] D. Beyer and M. Dangl. 2016. Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses. In *Proc. CAV (2) (LNCS 9780)*. Springer, 502–509. https://doi.org/10.1007/978-3-319-41540-6_28

[23] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

[24] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. 2020. Reproduction Package for TOSEM Article 'Verification Witnesses'. Zenodo. https://doi.org/10.5281/zenodo.3731856

[25] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. FSE*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867

[26] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. 2018. Tests from Witnesses: Execution-Based Validation of Verification Results. In *Proc. TAP (LNCS 10889)*. Springer, 3–23. https://doi.org/10.1007/978-3-319-92994-1_1

[27] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-Induction with Continuously-Refined Invariants. In *Proc. CAV (LNCS 9206)*. Springer, 622–640. https://doi.org/10.1007/978-3-319-21690-4_42

[28] D. Beyer, M. Dangl, and P. Wendler. 2018. A Unifying View on SMT-Based Software Verification. *J. Autom. Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[29] D. Beyer and K. Friedberger. 2020. Violation Witnesses and Result Validation for Multi-Threaded Programs. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 449–470. https://doi.org/10.1007/978-3-030-61362-4_26

[30] D. Beyer, S. Gulwani, and D. Schmidt. 2018. Combining Model Checking and Data-Flow Analysis. In *Handbook of Model Checking*. Springer, 493–540. https://doi.org/10.1007/978-3-319-10575-8_16

[31] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The Software Model Checker Blast. *Int. J. Softw. Tools Technol. Transfer* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z

[32] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *Proc. FSE*. ACM, Article 57, 11 pages. https://doi.org/10.1145/2393596.2393664

[33] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. 2007. Path Invariants. In *Proc. PLDI*. ACM, 300–309. https://doi.org/10.1145/1250734.1250769

[34] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[35] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. 2013. Information Reuse for Multi-goal Reachability Analyses. In *Proc. ESOP (LNCS 7792)*. Springer, 472–491. https://doi.org/10.1007/978-3-642-37036-6_26

[36] D. Beyer and M.-C. Jakobs. 2019. CoVeriTest: Cooperative Verifier-Based Testing. In *Proc. FASE (LNCS 11424)*. Springer, 389–408. https://doi.org/10.1007/978-3-030-16722-6_23

[37] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. 2018. Reducer-Based Construction of Conditional Verifiers. In *Proc. ICSE*. ACM, 1182–1193. https://doi.org/10.1145/3180155.3180259

[38] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proc. CAV (LNCS 6806)*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[39] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197. https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf

[40] D. Beyer and T. Lemberger. 2016. Symbolic Execution with CEGAR. In *Proc. ISoLA (LNCS 9952)*. Springer, 195–211. https://doi.org/10.1007/978-3-319-47166-2_14

[41] D. Beyer and T. Lemberger. 2019. TestCov: Robust Test-Suite Execution and Coverage Measurement. In *Proc. ASE*. IEEE, 1074–1077. https://doi.org/10.1109/ASE.2019.00105

[42] D. Beyer and S. Löwe. 2013. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE (LNCS 7793)*. Springer, 146–162. https://doi.org/10.1007/978-3-642-37057-1_11

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

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

[45] D. Beyer and H. Wehrheim. 2020. Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 143–167. https://doi.org/10.1007/978-3-030-61362-4_8

[46] D. Beyer and P. Wendler. 2012. Algorithms for Software Model Checking: Predicate Abstraction vs. Impact. In *Proc. FMCAD*. FMCAD, 106–113. https://www.sosy-lab.org/research/pub/2012-FMCAD.Algorithms_for_Software_Model_Checking.pdf

[47] D. Beyer and P. Wendler. 2013. Reuse of Verification Results: Conditional Model Checking, Precision Reuse, and Verification Witnesses. In *Proc. SPIN (LNCS 7976)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-39176-7_1

[48] P. Bielik, V. Raychev, and M. T. Vechev. 2017. Learning a Static Analyzer from Data. In *Proc. CAV (LNCS 10426)*. Springer, 233–253. https://doi.org/10.1007/978-3-319-63387-9_12

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

[50] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs. In *Proc. TACAS (LNCS 1579)*. Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[51] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. 2001. GraphML Progress Report. In *Graph Drawing (LNCS 2265)*. Springer, 501–512. https://doi.org/10.1007/3-540-45848-4_59

[52] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer. 2015. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In *Proc. FASE (LNCS 9033)*. Springer, 84–99. https://doi.org/10.1007/978-3-662-46675-9_6

[53] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. CCS*. ACM, 322–335. https://doi.org/10.1145/1180405.1180445

[54] H. Cai, Z. Shao, and A. Vaynberg. 2007. Certified self-modifying code. *ACM SIGPLAN Notices* 42, 6 (2007), 66–77. https://doi.org/10.1145/1250734.1250743

[55] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. 2015. Moving Fast with Software Verification. In *Proc. NFM (LNCS 9058)*. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[56] R. Castaño, V. A. Braberman, D. Garbervetsky, and S. Uchitel. 2017. Model Checker Execution Reports. In *Proc. ASE*. IEEE, 200–205. https://doi.org/10.1109/ASE.2017.8115633

[57] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. 2004. Modular Verification of Software Components in C. *IEEE Trans. Softw. Eng.* 30, 6 (2004), 388–402. https://doi.org/10.1109/TSE.2004.22

[58] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. 2016. The Kind 2 Model Checker. In *Proc. CAV (LNCS 9780)*. Springer, 510–517. https://doi.org/10.1007/978-3-319-41540-6_29

[59] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[60] E. M. Clarke, O. Grumberg, K. L. McMillan, and Xudong Zhao. 1995. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *Proc. DAC*. ACM, 427–432. https://doi.org/10.1145/217474.217565

[61] E. M. Clarke and H. Veith. 2003. Counterexamples Revisited: Principles, Algorithms, Applications. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday (LNCS 2772)*. Springer, 208–224. https://doi.org/10.1007/978-3-540-39910-0_9

[62] H. Cleve and A. Zeller. 2005. Locating Causes of Program Failures. In *Proc. ICSE*. ACM, 342–351. https://doi.org/10.1145/1062455.1062522

[63] B. Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Proc. CAV (2) (LNCS 10981)*. Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[64] P. Cousot and R. Cousot. 1976. Static determination of dynamic properties of programs. In *Proc. Int. Symp. on Programming*. Dunod, 106–130. https://www.di.ens.fr/~cousot/COUSOTpapers/publications.www/CousotCousot-ISOP-76-Dunod-p106--130-1976.pdf

[65] P. Cousot and R. Cousot. 1979. Systematic design of program-analysis frameworks. In *Proc. POPL*. ACM, 269–282. https://doi.org/10.1145/567752.567778

[66] W. Craig. 1957. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *J. Symb. Log.* 22, 3 (1957), 250–268. https://doi.org/10.2307/2963593

[67] C. Csallner and Y. Smaragdakis. 2005. Check 'n' crash: Combining static checking and testing. In *Proc. ICSE*. ACM, 422–431. https://doi.org/10.1145/1062455.1062533

[68] D. Dams and K. S. Namjoshi. 2005. Orion: High-Precision Methods for Static Error Analysis of C and C++ Programs. In *Proc. FMCO (LNCS 4111)*. Springer, 138–160. https://doi.org/10.1007/11804192_7

[69] A. F. Donaldson, L. Haller, D. Kröning, and P. Rümmer. 2011. Software Verification Using k-Induction. In *Proc. SAS (LNCS 6887)*. Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[70] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. 2010. Slab: A Certifying Model Checker for Infinite-State Concurrent Systems. In *Proc. TACAS (LNCS 6015)*. Springer, 271–274. https://doi.org/10.1007/978-3-642-12002-2_22

[71] E. Ermis, M. Schäf, and T. Wies. 2012. Error Invariants. In *Proc. FM (LNCS 7436)*. Springer, 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[72] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. 2011. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *Proc. ParCo (APC 22)*. IOS, 481–490. https://doi.org/10.3233/978-1-61499-041-3-481

[73] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz. 2018. Executable Counterexamples in Software Model Checking. In *Proc. VSTTE (LNCS 11294)*. Springer, 17–37. https://doi.org/10.1007/978-3-030-03592-1_2

[74] E. Ghassabani, A. Gacek, and M. W. Whalen. 2016. Efficient Generation of Inductive Validity Cores for Safety Properties. In *Proc. FSE*. ACM, 314–325. https://doi.org/10.1145/2950290.2950346

[75] J. Giesl, F. Mesnard, A. Rubio, R. Thiemann, and J. Waldmann. 2015. Termination Competition (termCOMP 2015). In *Proc. CADE (LNCS 9195)*. Springer, 105–108. https://doi.org/10.1007/978-3-319-21401-6_6

[76] P. Godefroid, N. Klarlund, and K. Sen. 2005. Dart: Directed Automated Random Testing. In *Proc. PLDI*. ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[77] S. Graf and H. Saïdi. 1997. Construction of Abstract State Graphs with Pvs. In *Proc. CAV (LNCS 1254)*. Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10

[78] A. Groce, S. Chaki, D. Kröning, and O. Strichman. 2006. Error Explanation with Distance Metrics. *STTT* 8, 3 (2006), 229–247. https://doi.org/10.1007/s10009-005-0202-0

[79] A. Groce and W. Visser. 2003. What Went Wrong: Explaining Counterexamples. In *Proc. SPIN (LNCS 2648)*. Springer, 121–135. https://doi.org/10.1007/3-540-44829-2_8

[80] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. 2006. Synergy: A new algorithm for property checking. In *Proc. FSE*. ACM, 117–127. https://doi.org/10.1145/1181775.1181790

[81] E. L. Gunter and D. A. Peled. 1999. Path Exploration Tool. In *Proc. TACAS (LNCS 1579)*. Springer, 405–419. https://doi.org/10.1007/3-540-49059-0_28

[82] A. K. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. 2008. Proving Non-Termination. In *Proc. POPL*. ACM, 147–158. https://doi.org/10.1145/1328438.1328459

[83] M. Heizmann, Y.-W. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. 2017. Ultimate Automizer with an On-Demand Construction of Floyd-Hoare Automata (Competition Contribution). In *Proc. TACAS (LNCS 10206)*. Springer, 394–398. https://doi.org/10.1007/978-3-662-54580-5_30

[84] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[85] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. 2002. Temporal-safety proofs for systems code. In *Proc. CAV*. Springer, 526–538. https://doi.org/10.1007/3-540-45657-0_45

[86] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. 2003. Extreme Model Checking. In *Verification: Theory and Practice*. Springer, 332–358. https://doi.org/10.1007/978-3-540-39910-0_16

[87] M. J. H. Heule. 2016. The DRAT format and DRAT-trim checker. *CoRR* 1610, 06229 (October 2016), 6 pages. https://arxiv.org/abs/1610.06229

[88] M. J. H. Heule. 2018. Schur Number Five. In *Proc. AAAI*. AAAI Press, 6598–6606. https://ojs.aaai.org/index.php/AAAI/article/view/12209

[89] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. 2010. How did you specify your test suite. In *Proc. ASE*. ACM, 407–416. https://doi.org/10.1145/1858996.1859084

[90] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. 2003. Data Flow Testing as Model Checking. In *Proc. ICSE*. IEEE, 232–243. https://doi.org/10.1109/ICSE.2003.1201203

[91] H. S. Hong, I. Lee, and O. Sokolsky. 2001. *Automatic Test Generation From Statecharts Using Modle Checking*. Technical Report MS-CIS-01-07. University of Pennsylvania. 27 pages. https://repository.upenn.edu/cgi/viewcontent.cgi?article=1092&context=cis_reports

[92] F. Howar and M. Mues. 2022. GWit (Competition Contribution). In *Proc. TACAS (2) (LNCS 13244)*. Springer.

[93] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. 2017. Safety Verification of Deep Neural Networks. In *Proc. CAV (LNCS 10426)*. Springer, 3–29. https://doi.org/10.1007/978-3-319-63387-9_1

[94] A. Iliasov. 2011. Generation of certifiably correct programs from formal models. In *Proc. WoSoCER*. IEEE, 43–48. https://doi.org/10.1109/WoSoCER.2011.14

[95] M.-C. Jakobs and H. Wehrheim. 2014. Certification for Configurable Program Analysis. In *Proc. SPIN*. ACM, 30–39. https://doi.org/10.1145/2632362.2632372

[96] M.-C. Jakobs and H. Wehrheim. 2017. Programs from Proofs: A Framework for the Safe Execution of Untrusted Software. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017), 7:1–7:56. https://doi.org/10.1145/3014427

[97] C. Jard and T. Jéron. 2005. TGV: Theory, Principles, and Algorithms. *STTT* 7, 4 (2005), 297–315. https://doi.org/10.1007/s10009-004-0153-x

[98] M. Jose and R. Majumdar. 2011. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *Proc. CAV (LNCS 6806)*. Springer, 504–509. https://doi.org/10.1007/978-3-642-22110-1_40

[99] T. Kahsai and C. Tinelli. 2011. PKind: A Parallel k-Induction Based Model Checker. In *Proc. Int. Workshop on Parallel and Distributed Methods in Verification (EPTCS 72)*. EPTCS, 55–62. https://doi.org/10.4204/EPTCS.72.6

[100] A. V. Khoroshilov, V. S. Mutilin, A. K. Petrenko, and V. Zakharov. 2009. Establishing Linux Driver Verification Process. In *Proc. Ershov Memorial Conference (LNCS 5947)*. Springer, 165–176. https://doi.org/10.1007/978-3-642-11486-1_14

[101] G. A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proc. POPL* (Boston, Massachusetts). ACM, 194–206. https://doi.org/10.1145/512927.512945

[102] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. 2006. Introducing the Open Trace Format (OTF). In *Proc. ICCS (LNCS 3992)*. Springer, 526–533. https://doi.org/10.1007/11758525_71

[103] D. Kröning and N. Sharygina. 2005. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *Proc. MEMOCODE*. IEEE, 101–110. https://doi.org/10.1109/MEMCOD.2005.1487900

[104] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. 2007. Efficient Unit Test Case Minimization. In *Proc. ASE*. ACM, 417–420. https://doi.org/10.1145/1321631.1321698

[105] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. 2012. Residual investigation: predictive and precise bug detection. In *Proc. ISSTA*. ACM, 298–308. https://doi.org/10.1145/2338965.2336789

[106] R. Majumdar and K. Sen. 2007. Hybrid Concolic Testing. In *Proc. ICSE*. IEEE, 416–426. https://doi.org/10.1109/ICSE.2007.41

[107] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. 2011. Certifying Algorithms. *Computer Science Review* 5, 2 (2011), 119–161. https://doi.org/10.1016/j.cosrev.2010.09.009

[108] K. L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Proc. CAV (LNCS 2725)*. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[109] K. L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proc. CAV (LNCS 4144)*. Springer, 123–136. https://doi.org/10.1007/11817963_14

[110] P. Müller and J. N. Ruskiewicz. 2011. Using Debuggers to Understand Failed Verification Attempts. In *Proc. FM (LNCS 6664)*. Springer, 73–87. https://doi.org/10.1007/978-3-642-21437-0_8

[111] K. S. Namjoshi. 2001. Certifying Model Checkers. In *Proc. CAV (LNCS 2102)*. Springer, 2–13. https://doi.org/10.1007/3-540-44585-4_2

[112] G. C. Necula. 1997. Proof-Carrying Code. In *Proc. POPL*. ACM, 106–119. https://doi.org/10.1145/263699.263712

[113] F. Nielson, H. R. Nielson, and C. Hankin. 1999. *Principles of Program Analysis*. Springer. https://doi.org/10.1007/978-3-662-03811-6

[114] Evgeny Novikov and Ilja S. Zakharov. 2017. Towards Automated Static Verification of GNU C Programs. In *Proc. PSI (LNCS 10742)*. Springer, 402–416. https://doi.org/10.1007/978-3-319-74313-4_30

[115] OASIS. 2019. Static Analysis Results Interchange Format (SARIF) Version 2.0. https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.html

[116] H. Ponce-De-Leon, T. Haas, and R. Meyer. 2022. Dartagnan: SMT-based Violation Witness Validation (Competition Contribution). In *Proc. TACAS (2) (LNCS 13244)*. Springer.

[117] H. O. Rocha, R. S. Barreto, L. C. Cordeiro, and A. Dias Neto. 2012. Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples. In *Proc. IFM (LNCS 7321)*. Springer, 128–142. https://doi.org/10.1007/978-3-642-30729-4_10

[118] F. B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

[119] K. Sen, D. Marinov, and G. Agha. 2005. Cute: A Concolic Unit Testing Engine for C. In *Proc. FSE*. ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[120] O. Ŝerý. 2009. Enhanced Property Specification and Verification in Blast. In *Proc. FASE (LNCS 5503)*. Springer, 456–469. https://doi.org/10.1007/978-3-642-00593-0_32

[121] C. Sternagel and R. Thiemann. 2014. The Certification Problem Format. In *Proc. UITP (EPTCS 167)*. EPTCS, 61–72. https://doi.org/10.4204/EPTCS.167.8

[122] J. Švejda, P. Berger, and J.-P. Katoen. 2020. Interpretation-Based Violation Witness Validation for C: NitWit. In *Proc. TACAS (LNCS 12078)*. Springer, 40–57. https://doi.org/10.1007/978-3-030-45190-5_3

[123] A. Taleghani and J. M. Atlee. 2010. Search-Carrying Code. In *Proc. ASE*. ACM, 367–376. https://doi.org/10.1145/1858996.1859079

[124] A. Turing. 1949. Checking a Large Routine. In *Report on a Conference on High Speed Automatic Calculating Machines*. Cambridge Univ. Math. Lab., 67–69. http://dl.acm.org/citation.cfm?id=94938.94952

[125] W. Visser, C. S. Păsăreanu, and S. Khurshid. 2004. Test-Input Generation with Java PathFinder. In *Proc. ISSTA*. ACM, 97–107. https://doi.org/10.1145/1007512.1007526

[126] M. Wagner, A. Knüpfer, and W. E. Nagel. 2016. OTFX: An In-memory Event Tracing Extension to the Open Trace Format 2. In *Proc. ICA3PP (LNCS 10049)*. Springer, 3–17. https://doi.org/10.1007/978-3-319-49956-7_1

[127] T. Wahl. 2013. The k-Induction Principle. http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf

[128] N. Wetzler, M. J. H. Heule, and Warren A. Hunt Jr. 2014. Drat-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Proc. SAT (LNCS 8561)*. Springer, 422–429. https://doi.org/10.1007/978-3-319-09284-3_31

[129] M. Whalen, J. Schumann, and B. Fischer. 2002. Synthesizing Certified Code. In *Proc. FME*. Springer, 431–450. https://doi.org/10.1007/3-540-45614-7_25

[130] T. Wu, P. Schrammel, and L. Cordeiro. 2022. Wit4Java: A violation-witness validator for Java Verifiers (Competition Contribution). In *Proc. TACAS (2) (LNCS 13244)*. Springer.

[131] A. Zeller. 2002. Isolating Cause-Effect Chains from Computer Programs. In *Proc. FSE*. ACM, 1–10. https://doi.org/10.1145/587051.587053