✓ Artifact evaluated by FSE

Correctness Witnesses: Exchanging Verification Results between Verifiers

Dirk Beyer [1], Matthias Dangl [2], Daniel Dietsch [3], Matthias Heizmann [3]

[1] LMU Munich, Germany [2] University of Passau, Germany [3] University of Freiburg, Germany

ABSTRACT

Standard verification tools provide a counterexample to witness a specification violation, and, since a few years, such a witness can be validated by an independent validator using an exchangeable witness format. This way, information about the violation can be shared across verification tools and the user can use standard tools to visualize and explore witnesses. This technique is not yet established for the correctness case, where a program fulfills a specification. Even for simple programs, it is often difficult for users to comprehend why a given program is correct, and there is no way to independently check the verification result. We close this gap by complementing our earlier work on violation witnesses with correctness witnesses. While we use an extension of the established common exchange format for violation witnesses to represent correctness witnesses, the techniques for producing and validating correctness witnesses are different. The overall goal to make proofs available to engineers is probably as old as programming itself, and proof-carrying code was proposed two decades ago — our goal is to make it practical: We 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. At any time, the invariants from the correctness witness can be used to reconstruct a correctness proof to establish trust. We extended two state-of-the-art verifiers, CPAchecker and UltimateAutomizer, to produce and validate witnesses, and report that the approach is promising on a large set of verification tasks.

CCS Concepts

•Software and its engineering → Formal software verification;

Keywords

Correctness Witness, Witness Validation, Software Verification, Program Analysis, Model Checking

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950351

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) [1] [1, 2, 3], many successful software verifiers were developed.

Recently, the problem of false alarms that verification tools sometimes produce has been addressed [5]: Formerly, a verification tool reported found bugs as counterexample traces in a tool-specific manner; those counterexamples were often not readable and therefore hardly usable. Determining whether the reported bug was a false alarm or described an actual programming error that needed to be fixed was a tedious manual process for the user. Exchangeable violation witnesses resolve this issue, because the general syntax allows new tools for presentation to be developed and used [4]. Witnesses should be considered as first-class objects that have much more value than the actual verification result TRUE or FALSE. A verification result should be trusted only if the *reason* for the result is provided, and the result can be re-established with the additional information. The process of witness validation is fully automatic.

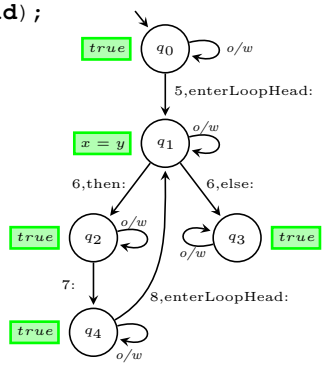
This paper complements our previous work on violation witnesses [5] by a method for producing and validating *correctness witnesses*. The most recent edition of the TACAS International Competition on Software Verification [3] revealed that soundness is a big issue: ten out of 13 participating verifiers in the category 'Overall' reported wrong correctness claims for verification tasks with known specification violations. One of the submissions was even claiming safety for 962 out of 2 348 verification tasks that were known to contain a bug. This rather embarrassing situation of the state-of-the-art in software verification can be fixed by producing correctness witnesses and letting a witness validator confirm the result. The result should be trusted only if it can be confirmed by at least one other verifier.

We propose that a verifier should be required to augment a verification result with a machine-readable and exchangeable witness, such that both, bug alarms and claims of safety, may be validated. With this technique, a trusted validator establishes trust in the verification results produced by an untrusted verifier, and even in the absence of a trusted validator the user's confidence in a verification result can be

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

```
1  extern unsigned int nondet(void);
2
3  int main() {
4    unsigned int x = nondet();
5    unsigned int y = x;
6    while (x < 1024) {
7      x = x + 1;
8      y = y + 1;
9    }
10   // Valid safety property
11   if (x != y) {
12     ERROR: return 1;
13   }
14   return 0;
15 }
```

(a) Safe program



(b) Witness automaton

```
1  extern unsigned int nondet(void);
2
3  int main() {
4    unsigned int x = nondet();
5    unsigned int y = x;
6    while (x < 1024) {
7      x = x + 1;
8      y = x + 1; // Bug
9    }
10   // Invalid safety property
11   if (x != y) {
12     ERROR: return 1;
13   }
14   return 0;
15 }
```

(c) Unsafe program

Figure 1: Example C programs (a,c) and a potential correctness witness (b)

increased by applying different validators to a verification witness. Witnesses can be read by humans (perhaps using a visualization or inspection tool) or by a witness validator.

This paper reports our experience with implementing two different witness-producing verifiers and two different witness validators for correctness witnesses. On the syntactic level, we use XML, more specifically GraphML [12], as a language to represent correctness witnesses. On the semantic level, we use the standard concept of (non-deterministic) finite automata to represent correctness witnesses. A correctness-witness automaton observes the program locations (along the control flow) that the verifier explores and provides invariants that hold at the locations that the verifier visits. A correctness witness is valid if its predicates are invariants for the program, and a validator should reject witnesses with incorrect invariants. The strength of the invariants determines the quality of the witnesses, but no particular strength is required. Witness 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.

Example. We illustrate the idea of correctness witnesses using two short C programs and an example witness automaton. The first of the two C programs is listed in Fig. 1a. It is taken from the category *Loops* of the benchmark set of the TACAS International Competition on Software Verification [2] [3]. It contains two unsigned integer variables `x` and `y`. Variable `x` is initialized to a non-deterministic value in line 4, and `y` is set to the value of `x` in line 5. Lines 6–9 contain a `while` loop that increments both variables in each iteration while the value of `x` is less than 1 024. In the lines 10–13, the safety property is asserted, which requires that `x` equals `y`. While the safety property trivially holds before the first execution of the loop body, a verifier has to find out that $x = y$ is a loop invariant in order to prove that the safety property holds after the loop. Because the loop invariant $x = y$ is *inductive*, it is easy to prove that it holds. Therefore, proving the whole program correct is easy if the loop invariant $x = y$ is given, but finding such a loop invariant is in general hard, and depends on the employed verification strategy: it is the critical step in verifying this program.

A verifier that successfully proves the safety property for the program may then export a correctness witness. If the correctness witness contains the invariant $x = y$, a witness validator should be able to easily confirm the correctness witness. Figure 1b displays a graphical representation of such a correctness witness, which is actually produced by our implementation in CPAchecker for the program listed in Fig. 1a (we reduced it to the most important parts for readability). Our implementation in UltimateAutomizer produces a very similar witness. The witness uses the same type of syntactic guards as the violation witnesses that we introduced in a previous work [5], to match automaton transitions with program operations. The automaton starts in an initial control state q_0. The witness assigns the invariant *true* to control state q_0. It is allowed to proceed to state q_1 if the control-flow enters the loop head. As long as this transition is not possible, the automaton remains in state q_0 via the self-transition 'otherwise' (`o/w`). From q_1, the automaton can proceed to state q_2 if the condition of the `while` loop in line 6 is true (the `then`-case), or to state q_3 if the condition in line 6 is false (the `else`-case). As long as none of these transitions are possible, the automaton remains in state q_1 via the self-transition `o/w`. The automaton proceeds from state q_2 to q_4 on the program operation in line 7 and from there back to state q_1 after the program operation in line 8. As long as these conditions are not met, the automaton will stay in each of the control states via their self-transitions `o/w`. If the automaton is in state q_3, it will stay there forever.[3] States q_0, q_2, q_3, and q_3 contain the trivial invariant *true*. State q_1 specifies the invariant $x = y$. Because state q_1 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 witness.

The second program in Fig. 1c is almost identical to the program in Fig. 1a, with the critical difference that there

[2] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp16/c/loop-acceleration/multivar_true-unreach-call1.i

[3] 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.

is a bug in line 8, which causes the safety property to be violated: After each iteration, the value of `y` equals `x + 1` instead of `x`. Due to the structural similarity between the two programs, the witness in Fig. 1b can also be matched with the second (unsafe) program. A validator that checks the loop invariant $x = y$ will fail to prove its invariance, and thus will reject the witness. Because each validation of a correctness witness also implicitly uses the safety property as an invariant, the validator could alternatively also reject the witness by finding a counterexample to the safety property in lines 10–13 before disproving the loop invariant $x = y$.

Related Work. The exchange format for correctness witnesses and the corresponding techniques for communicating verification witnesses across verification tools are based on previous work on violation witnesses [5]. While the tasks of producing and validating correctness witnesses as well as the involved concepts are different from those for violation witnesses, we were able to reuse the exchange format for violation witnesses with only minor extensions, namely adding tags for location invariants and a new syntactic guard to identify loop heads. The stability of the format is important because it may incite developers of other verifiers (which perhaps already support violation witnesses) to support correctness witnesses as well. Analogous to the concept of stepwise testification of violation witnesses, a correctness-witness validator becomes a correctness-witness testifier if the validator itself documents its reasoning again in a correctness witness. Before the common exchange format, violation witnesses were used only based on proprietary formats within particular tools. For example, ESBMC was extended to reproduce errors via instrumented code [27], and CPACHECKER was used to re-check previously computed error paths by interpreting them as automata that control the state-space search [11]. The competition on termination uses CPF [28] to store termination proofs for term-rewrite systems.

Proof-Carrying Code (PCC). Ideas on proof witnesses have previously been explored in the context of proof-carrying code [26]. PCC is a mechanism where an untrusted source supplies both an executable program and a proof witness that can be checked against the program and specification by a trusted validator to establish trust in the program. Intermediate results during the verification procedure can be used by certifying model checkers to compose and dump full proofs as proof certificates [25].

The two implementations of correctness-witness validation that we provide and the presented exchange format for correctness witnesses enable the mechanism of proof-carrying code for real-world C programs and allow further verification tools to adopt the technique. The main difference of our work to proof-carrying code is that we do not strictly require the witness to contain a full proof. We found that in practice, a full proof for even small programs may become very large in size unless a considerable amount of additional effort is spent on simplifying formulas. Especially for larger programs, it is often neither wanted nor even feasible to deal with such a full proof — as in math, good lemmas or proof sketches are of essence. Therefore, we support flexibility: the better the witness, the more likely it is that the witness validator will quickly confirm it; a less detailed witness may also succeed in guiding the validator to the proof, but in turn may require more effort from the validator. In addition, we do not use the program to carry the proof, but consider the witness separately as an own first-class object (separation of concerns, flexibility, maintainability).

Certificates. Correctness certificates have long been used for increasing the trust in code generated from some form of formal description or model (e.g. [13, 14, 18, 20, 31]). Those correctness certificates are complete proofs of functional correctness. Our exchange format can also be used as correctness certificate and to represent a full proof, but this is not required: a correctness witness can —more generally— be a partial proof of correctness.

Reusing Reachability Graphs. Many model checkers materialize the intermediate results produced by their state-space exploration as an abstract reachability graph (ARG). The ARG, which is the basic data structure in tools like SLAM, BLAST, and CPACHECKER, can be used to extract invariants of the program [18], which in turn can be used for PCC, or for extreme model checking [19], which checks if a previously computed ARG is a safety proof for the given (slightly changed) input program. SLAB [15] is a certifying model checker that produces a proof certificate for the abstract model of a program in SMT-LIB format. Such a certificate can easily be checked using an SMT solver, but mapping it back to the original program to validate that it indeed certifies the correctness *of the original program* is non-trivial. As a result, the user can only assume that the certificate faithfully refers to the original program.

Search-Carrying Code (SCC). The approach SCC [29] uses search scripts to guide a model checker along paths of the ARG. Search scripts and correctness witnesses share the important idea of guiding a validator through the state space in order to reconstruct the correctness proof. Search scripts can be seen as a special case of correctness witnesses where the invariants are omitted (witnesses support branching as well). Correctness witnesses overcome three limitations of search scripts (cf. Sect. 4.3 in [29]): (i) the exchange format is independent from the verification approach (not bound to explicit-state model checking), (ii) the approach works across different verifiers, even if built on different technologies (as shown in our evaluation with CPACHECKER and AUTOMIZER), and (iii) the approach allows a flexible mapping from program operations to the verifier-specific states and transitions that is more tolerant to code reduction (which was already used by violation witnesses and is supported by many verification tools [3]). We found these extensions are essential for practical impact.

Proof Programs and Configurable Certification. One aspect of PCC is the idea that validation should be much faster than verification. In programs-from-proofs [23], correctness certificates take the form of new, behaviorally equivalent programs that are generated by a predicate analysis. Those new programs can then be efficiently verified by a data-flow analysis alone, although they may be exponentially larger in terms of lines of code. As their control-flow is necessarily different, they may exhibit completely different run-time behavior. Certificates for configurable program analysis [21, 22] represent all reachable states of a program as correctness certificate, which is comparable to a correctness witness with an invariant for each program location. Then, in order to speed up the validation, various size-reducing techniques are applied. Because correctness witnesses can contain partial proofs, a validator may choose to complement a partial proof with its own verification strategy or even perform the full

verification of a verification task itself; the validator never assumes that a correctness witness constitutes a complete proof. Therefore, the validation of these witnesses does not consistently exhibit a speedup. Nevertheless, similar techniques can be applied if one assumes that the witnesses represent complete proofs. Because both implemented witness producers, CPAchecker and Automizer, restrict themselves to loop invariants and procedure post conditions, the size of the witnesses is not an issue.

Partial Verification. Verifiers have three possible outcomes: a verifier either (1) finds a bug, (2) proves correctness, or (3) fails. Error witnesses [5] and correctness witnesses improve the first and second case, respectively. Conditional model checking (CMC) [7] improves the third case, by advocating reports of partial verification results. The output condition describes the result of an incomplete verification attempt (which part of the state was successfully verified), and the input condition instructs a model checker to only partially verify a system (which part of the state space is to be verified). Subsequent verification runs with a different approach can be used to complete the verification. Witnesses can be used to complement CMC by describing (a) invariants (in correctness witnesses) that were used to verify the part of the system that was successfully verified and (b) paths (in violation witnesses) that hindered a complete verification.

2. CORRECTNESS WITNESSES

The goal of our work is to represent verification results in such a way that they are reproducible, machine-readable, and exchangeable between different verifiers. This paper focuses on correctness witnesses, i.e., witnesses that provide evidence that the given program satisfies the given specification. We use witness automata to represent witnesses.

For the theoretical foundation of our work, we refer the reader to our concepts for violation witnesses [5]. Here, we only explain the difference and give an informal motivation. We restrict our presentation to a simple imperative programming language that contains only assignment and assume operations, and where all program variables are integers. Our implementations are based on CPAchecker [9] and UltimateAutomizer [16], both of which support C programs. We use control-flow automata (CFA) to represent programs. A *control-flow automaton* consists of a set of program locations (modelling the program counter), the initial program location (program entry), and a set of control-flow edges, each of which models an operation that is executed during the flow of control from one program location to another.

A *correctness-witness automaton* is an observer automaton, and a *correctness-witness analysis* can be formalized using the notion of configurable program analysis (CPA) [8], resulting in an observer CPA for a correctness-witness automaton, which runs as one component CPA of a composite program analysis in parallel to other component CPAs. In contrast to the control automata that we use for violation witnesses [5], correctness-witness automata do not restrict the exploration of the program's state space but only observe the state-space exploration. 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. The program analysis of a witness validator checks if the given invariants indeed hold at their corresponding abstract program states; a witness is rejected if the predicate ϕ for an abstract program state is not confirmed.

There are only two differences between violation witnesses and correctness witnesses:

- A violation witness has assumptions at the witness automaton's transitions that restrict the state space; a correctness witness does not restrict the state space but contains a state invariant at each control state in the witness automaton.
- A validator for a violation witness tries to replay an error path through the program, while a validator for a correctness witness tries to replay the correctness proof, i.e., checks for each invariant whether all reachable program states are covered by the invariant and that no error state is contained in the state space that the invariants define.

Exchange Format for Correctness Witnesses. Our exchange format for correctness witnesses is an extension of our earlier format for violation witnesses [5]. The format is based on GraphML [12], where a graph contains nodes representing states and edges representing transitions of a witness automaton. The format supports adding attributes to the states, for example to mark them as initial state or error state, or to annotate invariants to a state. Transitions can also be labeled with *guard* attributes, like line numbers and assumptions, which can be used by witness validators to match the witness to the program and, for violation witnesses, to constrain the state-space of the program. In order to be able to express correctness witnesses, we extend the format as follows:

- Graphs can now be labeled with the attribute `witness-type`, for which `correctness_witness` or `violation_witness` are valid values. If omitted, the witness is assumed to be a violation witness.
- States can be labeled with the additional attribute `invariant`, which has to be a Boolean C expression that is valid in the scope given by `invariant.scope`. The attribute `invariant` represents an invariant that is required to hold at the program location that is represented by the state.
- If the witness type is `correctness_witness`, transitions are not allowed to have the attribute `assume`, because correctness-witness automata are observer automata, which do not restrict the search space of the program.
- If the witness type of is `correctness_witness`, marking states as violation states with the attribute `violation` is not allowed, because that would contradict the intention of the correctness witness.
- The syntactic guard `enterLoopHead` matches automaton states to loop heads in the control flow.

The decision to restrict `invariant` attributes to C expressions is based on the idea that it should be as easy as possible to extend an existing verification tool for C such that it can produce or validate correctness witnesses. Nevertheless, C is not suitable to express all aspects of invariants that are commonly used, like quantifiers, references to the return value of functions, or relations between variable values spanning several stack frames. Formal specification languages like

ACSL [4] support those constructs and would be a natural extension to the format, but were too complex for the first step towards exchangeable correctness witnesses, as they would likely hinder the quick adoption of the format by a wide range of verifiers.

3. CONSTRUCTION OF CORRECTNESS WITNESSES

There are many different ways to obtain program invariants, and different approaches give invariants of different quality. The better the invariants, the easier it is to understand the proof, and the more efficient it is to re-verify the program. We implemented two approaches, one based on k-induction in CPAchecker, and one based on automata in UltimateAutomizer.

3.1 CPAchecker's Verifier

The CPAchecker-based verifier that we extended to generate correctness witnesses uses the k-induction technique KI⇜DF [6]. k-induction combines techniques from bounded model checking with induction, in order to obtain unbounded safety proofs. Consider a candidate invariant P for a verification task that contains an unbounded loop. A bounded model check with bound $k = 1$ is able to show that no program path of length $k = 1$ exists for which P is violated (a), but it cannot prove the absence of longer counterexample paths. If P is inductive, i.e., for any given iteration through the loop where P holds before, P also holds after the iteration (b), induction can be used to prove that P is an invariant, taking (a) as the base case for the induction and (b) as the inductive step case.

For k-induction, this procedure is extended to larger values of k by asserting the invariant P for not only one but k consecutive predecessors in the step case. For $k > 1$, $(k-1)$-inductiveness implies k-inductiveness, therefore, k-induction may in practice be easier (because more constrained) to prove than $(k-1)$-induction [30]. Naturally, this procedure cannot succeed if P is not k-inductive for any k. For these cases, it is desirable to strengthen P with auxiliary invariants to try making the assertion inductive. In the k-induction technique KI⇜DF, an auxiliary-invariant generator (based on data-flow analysis) runs in parallel to the k-induction procedure and provides invariants to strengthen the induction hypothesis. As time progresses, the precision used by the invariant generator is increased, causing stronger invariants to be generated, until the auxiliary invariants sufficiently strengthen the induction hypothesis to become inductive, and the induction proof of the invariant P (which is the safety property) in conjunction with the auxiliary invariants succeeds. If the proof succeeds and the correctness witness is constructed, the auxiliary invariants that were used to strengthen the induction hypothesis are also attached to the respective location in the witness.

3.2 UltimateAutomizer's Verifier

Automizer follows an automata-based verification approach [17] in which a correctness proof is a sequence of automata. In this subsection we present this verification approach and demonstrate how we transform a correctness proof given as a sequence of automata into a correctness proof given as invariants.

[4]http://frama-c.com/download/acsl.pdf

In a first step, Automizer transforms the given program and the given correctness specification into a CFA with error locations. We consider this CFA as an acceptor of a formal language whose alphabet Σ is the set of all program operations and whose accepting states are the error locations of the program. The words accepted by this automaton are exactly the labelings of all paths that lead from the initial location to an error location. We say that a sequence of operations (a word over this alphabet) is *infeasible* if it does not correspond to any program execution. The analysis is based on the fact that the program is correct if and only if each word accepted by the CFA is an infeasible sequence of operations. During the verification process, Automizer iteratively constructs automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ over the alphabet Σ such that each automaton accepts only words that are infeasible. As soon as the union of the languages of these automata is a superset of the language accepted by the CFA, the verification process is finished and the automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ constitute a correctness proof for the program.

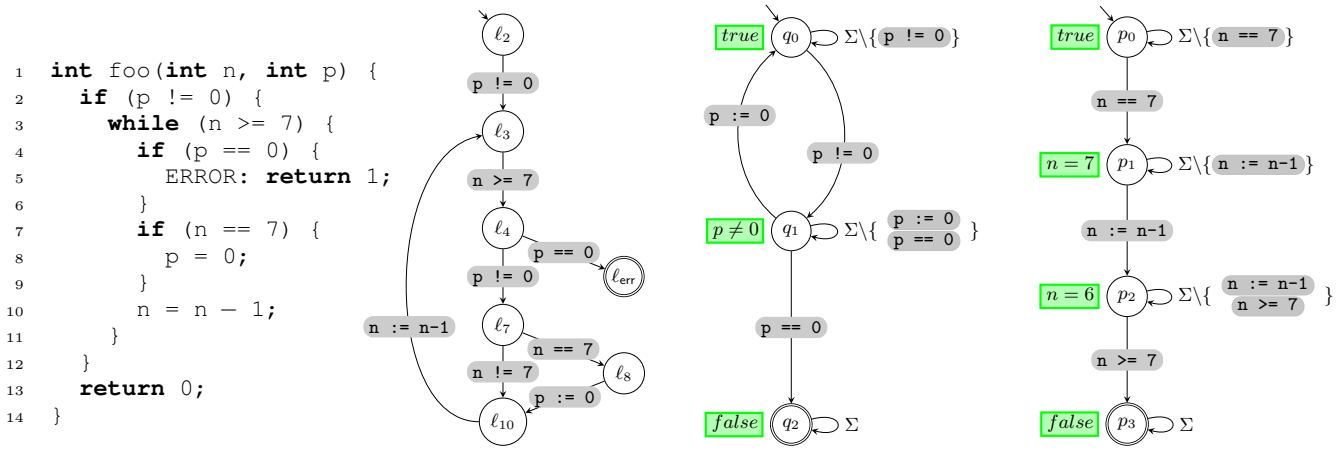
An example for such a proof is depicted in Fig. 2. The program on the left contains its correctness specification in the source code (lines 4–6), which states that the value of `p` is not 0. The program is correct. An intuitive argument to justify the correctness is that `p` can be set to 0 only in the very last iteration of the `while` loop. Automizer first translates this program into the CFA depicted in Fig. 2b. We note that for this CFA we applied an optimization that removes all nodes from which there is no path to an error location. Next, Automizer constructs the two automata $\mathcal{A}_1$ (Fig. 2c) and $\mathcal{A}_2$ (Fig. 2d) as a correctness proof for the program. The automaton $\mathcal{A}_1$ accepts all sequences of operations that reach the error location but did not take the `if` branch (line 7) in the preceding iteration. All these sequences of operations are infeasible because the operation `p != 0` and the operation `p == 0` are contradicting each other. The automaton $\mathcal{A}_2$ accepts all sequences of operations that take the `if` branch and enter the `while` loop another time. All these sequences of operations are infeasible, because the operations `n == 7`, `n := n-1` and `n >= 7` cannot be executed after each other.

Automizer uses the concept of *Floyd-Hoare automata* [17] to construct the automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ that constitute the correctness proof. A Floyd-Hoare automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, Q_{\mathsf{fin}})$ is an automaton over the alphabet Σ of the program's operations together with a mapping that assigns to each state $q \in Q$ a formula φ_q that denotes a predicate over the program variables such that the following holds:

- The initial state is annotated by the formula $true$.
- For each of the automaton's transitions $(q, op, q') \in \delta$, the triple $\{\varphi_q\}\ op\ \{\varphi_{q'}\}$ is a valid Hoare triple.
- Each accepting state is annotated by the formula $false$.

Hence, a Floyd-Hoare automaton accepts only sequences of operations that are infeasible. The automata depicted Fig. 2c and Fig. 2d are Floyd-Hoare automata. The formulas that are annotated to the automata's states are framed by cornered boxes. E.g., for the automaton $\mathcal{A}_1$, the state q_1 is annotated by the formula $p \neq 0$.

Given a CFA $\mathcal{A}_P$ and Floyd-Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ we use the following approach to construct invariants. In a first step, we construct an automata-theoretical product of the automata $\mathcal{A}_P$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n$. The states of the product are tuples of the form $(\ell, s_1, \ldots, s_n)$ where the first component (a state of the CFA) is a location of the program



(a) C code (b) Control-flow automaton (c) Floyd-Hoare automaton $\mathcal{A}_1$ (d) Floyd-Hoare automaton $\mathcal{A}_2$

Figure 2: Program whose correctness is specified by an error label (a); corresponding CFA (b); automata (c) and (d) represent internal correctness proof by AUTOMIZER; we construct a product of these automata in order to obtain invariants for the program

Table 1: Reachable states in the product automaton of $\mathcal{A}_P, \mathcal{A}_1$, and $\mathcal{A}_2$ together with their annotation

state	annotation	state	annotation
(ℓ_2, q_0, p_0)	$true \wedge true$	(ℓ_7, q_1, p_3)	$p \neq 0 \wedge false$
(ℓ_3, q_1, p_0)	$p \neq 0 \wedge true$	(ℓ_8, q_1, p_1)	$p \neq 0 \wedge n = 7$
(ℓ_3, q_0, p_2)	$true \wedge n = 6$	(ℓ_8, q_1, p_3)	$p \neq 0 \wedge false$
(ℓ_3, q_0, p_3)	$true \wedge false$	(ℓ_{10}, q_1, p_0)	$p \neq 0 \wedge true$
(ℓ_3, q_1, p_3)	$p \neq 0 \wedge false$	(ℓ_{10}, q_0, p_1)	$true \wedge n = 7$
(ℓ_4, q_1, p_0)	$p \neq 0 \wedge true$	(ℓ_{10}, q_0, p_3)	$true \wedge false$
(ℓ_4, q_0, p_3)	$true \wedge false$	(ℓ_{10}, q_1, p_3)	$p \neq 0 \wedge false$
(ℓ_4, q_1, p_3)	$p \neq 0 \wedge false$	(ℓ_{err}, q_2, p_0)	$false \wedge true$
(ℓ_7, q_1, p_0)	$p \neq 0 \wedge true$	(ℓ_{err}, q_0, p_3)	$p \neq 0 \wedge false$
		(ℓ_{err}, q_2, p_3)	$false \wedge false$

Table 2: Invariants for the program depicted in Fig. 2a

location	invariant
ℓ_2	$true$
ℓ_3	$p \neq 0 \vee n = 6 \vee false$
ℓ_4	$p \neq 0 \vee false$
ℓ_7	$p \neq 0 \vee false$
ℓ_8	$(p \neq 0 \wedge n = 7) \vee false$
ℓ_{10}	$p \neq 0 \vee n = 7 \vee false \vee false$
ℓ_{err}	$false \vee false \vee false$

and the $i+1$-th component s_i is a state of the automaton $\mathcal{A}_i$. We annotate each tuple in the product by a formula which is the n-ary conjunction of the annotations of all s_i, that is, the annotation of the tuple $(\ell, s_1, \ldots, s_n)$ is the conjunction $\bigwedge_{i=1}^{n} \varphi_{s_i}$. Table 1 shows the annotations for the reachable states in the product of the automata from Fig. 2.

In a second step, we obtain the invariant for a location ℓ by taking the disjunction of all annotations of all tuples that are reachable in the product and whose first component is location ℓ. Table 2 shows the invariants that we obtain for the program depicted in Fig. 2a.

In the current implementation of AUTOMIZER, we write the invariants only at loop heads into the witness file. The other invariants are constructed only optionally and are used for tool-internal sanity checks. For the program in Fig. 2a we output the invariant $p \neq 0 \vee n = 6$ at location ℓ_3.

4. VALIDATION OF CORRECTNESS WITNESSES

Out of the many possibilities to implement a witness validator, we show the potential and flexibility of the approach by describing two different strategies that are implemented in two different verification frameworks.

4.1 CPAchecker's Validator

Like the CPACHECKER-based verifier, the CPACHECKER-based validator for correctness witnesses uses 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. Then, a k-induction algorithm with an initial value $k = 1$ and iteratively increasing k is started as described in previous work [6]: For a given value of k, there is a bounded model-checking phase (the base case) followed by an induction phase (the inductive-step case). In the first phase, each invariant and the safety property are checked with a bounded model check with bound k. If the bounded model check detects a violation of the safety property, then the witness is rejected. If the bounded model check finds a counterexample to an invariant at some program location, then the invariant is removed from that program location, and, if no possible program location for this invariant remains, then the witness is rejected. In the second phase, each invariant and the safety property are checked for k-inductivity. If the safety property is k-inductive, then the program satisfies its specification and the witness validation terminates successfully. If an invariant is inductive, it is confirmed and can be used as a sound auxiliary invariant to strengthen future k-induction checks for the remaining unconfirmed invariants and the safety property. Then, k is incremented and the first phase starts again.

One of the benefits of using k-induction for witness validation is that k-induction for software verification is known to perform well on non-trivial verification tasks only if supplied with the necessary auxiliary invariants [6, 24]. All techniques that are implemented in CPACHECKER to generate its own aux-

iliary invariants are turned off for the validation. Therefore, the ability of this validator to prove that a given program satisfies its specification is tied to the quality of the given invariants in the witness.

Using the example program from Fig. 1a, the CPAchecker-based validator confirms the witness from Fig. 1b. If the invariant $x = y$ is removed from the witness for state q_1, the witness is still valid in principle (because *true* is an invariant). However, the k-induction-based validator will no longer confirm it, 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). If the CPAchecker-based validator is applied to the program from Fig. 1c, in which the safety property is violated, and the witness from Fig. 1b, the validator rejects the witness because it is able to find a counterexample for the invariant.

4.2 UltimateAutomizer's Validator

While validating a witness, Automizer verifies the given program and considers each invariant as an additional specification that has to be proven. For checking one of these specifications Automizer assumes that all other specifications are valid. The witness is confirmed if all specifications (including the original one) hold, otherwise the witness is rejected. The adding of the witness specifications is implemented as follows. First, the program that is given as a CFA is matched with the witness in order to determine which invariant belongs to which location. We match an invariant to a program location if an outgoing or incoming line from the invariant location is labeled with the same line number as the program location. If we have to match several invariants to the same location, we instead map their disjunction. The result of this first step is a partial map f from program locations to invariants. In a second step, we modify the CFA as follows. For each location ℓ for which the mapping f is defined, we add

- a new location ℓ',
- a new edge $(\ell, op_{f(\ell)}, \ell')$ where $op_{f(\ell)}$ is the assume operation that assumes the invariant $f(\ell)$ that was mapped to ℓ, and
- a new edge $(\ell, op_{\neg f(\ell)}, \ell_{\mathsf{err}})$, where $op_{\neg f(\ell)}$ is the assume operation which assumes the negation of the invariant $f(\ell)$ and ℓ_{err} is an error location.

Furthermore, we replace each outgoing edge of the form $(\ell, op, \ell")$ by an edge $(\ell', op, \ell")$. The resulting CFA is then verified as described in Sect. 3.2.

4.3 Testification

If the validation of a witness succeeds (i.e., the witness is confirmed), then CPAchecker and Automizer produce another correctness witness, which in turn contains all confirmed invariants. Therefore, the two validators that we implemented are not only consumers but also producers of correctness-witnesses. We call a witness validator that itself documents its process with another witness a witness testifier, based on the notion introduced for the corresponding concept for violation witnesses [5]. This feature is important for cases where the validator is untrusted. Several testifiers can then be chained together, such that even if the user does not trust any of the testifiers alone, a verdict supplemented by a witness that has been validated and potentially refined by testifiers that are implemented in different frameworks and based on different technology is less likely to be incorrect.

Witnesses that are produced by CPAchecker's validator are always at most as large as the witnesses used as input, because they contain at most all of the provided invariants, but may contain less if not all were required for the proof to succeed. Another application for this validator is therefore to compress witnesses by removing some irrelevant invariants, although it is not guaranteed to eliminate all irrelevant invariants. Also, this validator is idempotent with respect to the witnesses it produces, meaning that validating them again with the same validator will produce the same witness.

5. EXPERIMENTAL EVALUATION

To demonstrate the applicability of our approach, we performed a large number of experiments in a feasibility study. The experimental work flow consists of instructing the verifier (1) to produce a correctness witness and (2) to validate a correctness witness.

5.1 Experiment Goals

We define an exchange format for machine-readable witnesses to enable different verifiers to document the facts their proofs are based on in the form of correctness witnesses. We perform a feasibility study to support the following claims:

Claim 1: Witnesses produced by a verifier based on a certain framework can be validated by a validator based on the same framework, otherwise there is obviously an inconsistency in the communication of the invariants via the witnesses.

Claim 2: The correctness witnesses produced by a verifier based on one framework can be understood by a witness validator of a different framework.

Claim 3: The complexity of the validation of a correctness witness is related to the contents of the witness, i.e., there are verification tasks for which a verifier can produce witnesses such that the validation uses less resources to validate the witness than the verifier used to verify the verification task.

5.2 Benchmark Set

Our benchmark is composed of 3 411 verification tasks without any known specification violations from all categories of SV-COMP 2016 [3] except *ArraysMemSafety*, *HeapMemSafety*, *Recursive*, *Termination*, and *Concurrency*, which are not supported by the validator implemented in CPAchecker, or not supported by UltimateAutomizer. We also did not use the tasks from the demo category *BusyBox*, which has not been included as an official category by the jury of SV-COMP 2016 due to an apparent lack of quality of the contained tasks.

We considered including the verification tasks with known specification violations to check if correctness witnesses for wrong proofs are rejected, but CPAchecker did not produce any wrong proofs for these tasks. Of the four cases for which UltimateAutomizer produced wrong proofs for these tasks in SV-COMP 2016, we could only reproduce three[5] in

[5] The tasks are:
`ldv-linux-3.7.3/main4_false-unreach-call_drivers-scsi-mpt2sas-mpt2sas-ko-32_7a-linux-3.7.3.c`,
`43_2a_bitvector_linux-3.16-rc1.tar.xz-43_2a-drivers-scsi-megaraid-megaraid_`

the current version of UltimateAutomizer. The correctness witnesses that Automizer produces for these presumably incorrect proofs contain no candidate invariants, and the validation with CPAchecker was not able to confirm or reject these witnesses within a CPU time limit of 15 min in our experimental setup. Therefore, we cannot provide an extensive evaluation on the rejection of non-artificial known incorrect proofs.

5.3 Experimental Setup

Our experiments were conducted on machines with two 2.6 GHz 8-core CPUs (Intel Xeon E5-2650 v2) with 135 GB of RAM. The operating system was Ubuntu 16.04 (64 bit), using Linux 4.4 and OpenJDK 1.8. Each verification task was limited to two CPU cores, a CPU run time of 15 min, and a memory usage of 15 GB. We used version `cpachecker-1.6.8-fse16-correctnessWitnesses` of CPAchecker, with MathSAT5 as SMT solver. As a verifier, CPAchecker was configured to perform k-induction using the theory of bit-vector arithmetics and uninterpreted functions. The k-induction procedure was augmented by an auxiliary-invariant generator based on expressions over intervals. For the validator based on CPAchecker, the same configuration of k-induction as above for the verifier was used, but instead of synthesizing invariants, the validator uses only auxiliary invariants from the set of confirmed candidate invariants from the witness, as described in Sect. 4.1. UltimateAutomizer was used in revision `c3312191` from the `dev` branch, with Z3 as SMT solver. The benchmarks were executed using BenchExec [10] in version 1.9.

5.4 Presentation and Availability

The results, tools, and verification tasks that we used in our evaluation are available on the supplementary web page. [6] All reported times (CPU time) are rounded to two significant digits. Our knowledge about existing violations is based on the verdicts of the software-verification community.[7] If the validation of a witness exceeds its resource limits before confirming the witness, it is counted as a rejection.

5.5 Results

Claim 1: Consistency within the Same Framework. Our first experiment represents a feasibility study showing that we were able to implement a witness exchange format for correctness witnesses for C programs for CPAchecker and UltimateAutomizer, where both can take the roles of a verifier (producing witnesses) and a witness validator for their own witnesses. The first and last columns of Table 3 show that CPAchecker accepted 1 906 of 2 081 witnesses produced by CPAchecker, and that Automizer accepted 1 875 of 1 898 witnesses produced by Automizer, so that the acceptance rates for their own witnesses are 92 % and 99 %, respectively. Furthermore, for the rejected witnesses, Automizer detects incorrect invariants in seven of its own witnesses, and CPAchecker refutes invariants in four of its own witnesses [8]. We also performed an experiment where we applied correctness-witness testification by validating the witnesses *produced* by the CPAchecker-based witness validations (as mentioned in Sect. 4.3, our validators support testification [5]). In this experiment, the CPAchecker-based witness validator was able to confirm 1 905 of the 1 906 witnesses that it had produced during the validation of the witnesses produced by the CPAchecker-based verifier. We interpret the results for our first experiment as confirmation that the witnesses produced by both tools are consistent with their own frameworks.

Table 3: Accepted and Rejected Witnesses

Validator	CPAchecker		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer
Produced	2 081	1 898	2 081	1 898
Accepted	1 906	697	1 164	1 875
Rejected	175	1 201	917	23
Accept. rate	92 %	37 %	56 %	99 %

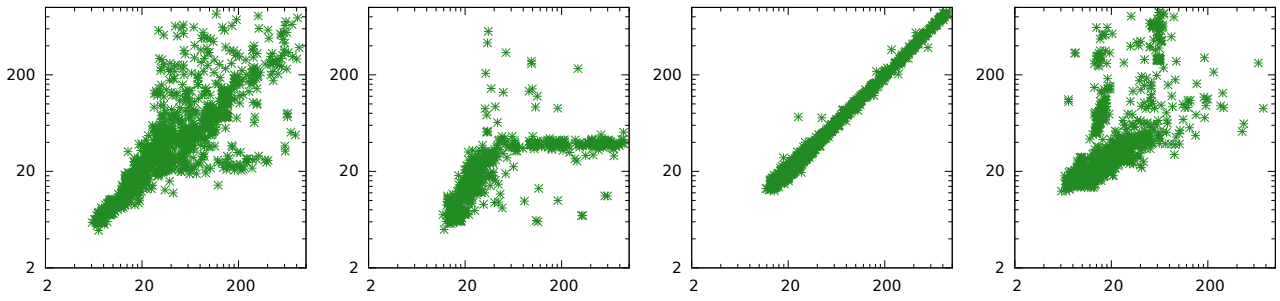
Claim 2: Validation across Frameworks. Our second experiment represents a feasibility study showing that we were able to communicate witnesses across frameworks, where witnesses produced by the CPAchecker-based verifier are validated by the Automizer-based validator and vice versa. Table 3 shows that CPAchecker accepted 37 % of the witnesses produced by Automizer, and that Automizer accepted 56 % of the witnesses produced by CPAchecker. These results are not yet as promising as those where the tools validate their own witnesses. We analyzed the rejections and found different causes for both cases: (1) CPAchecker did not detect any incorrect invariants in the witnesses produced by Automizer, and there are often too few invariants present in those witnesses for the k-induction-algorithm to succeed within the time limit. This means that CPAchecker mostly does not dispute the witnesses of Automizer, but it cannot confirm them either. (2) The prototypical implementation of the Automizer-based validator is not always able to find the correct program location for an invariant. If Automizer maps an invariant to the wrong program location, and the invariant does not hold there, the witness is rejected. While there is still room for improvement to our prototypical 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 shows that for between 700 to 1 200 of 1 900 to 2 100 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 witness, whereas previously, communicating such information between different tools was entirely impossible.

`mm.ko-entry_point_false-unreach-call.cil.out.c`, and `linux-4.2-rc1.tar.xz-43_2a-drivers-scsi-megaraid-megaraid_mm.ko-entry_point_false-unreach-call.cil.out.c`

[6] https://www.sosy-lab.org/research/correctness-witnesses/

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

[8] 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.



(a) CPAchecker / CPAchecker (b) Automizer / CPAchecker (c) Automizer / Automizer (d) CPAchecker / Automizer

Figure 3: Scatter plots for pairwise composition for witness validation: CPU seconds for producing a witness on the x axis, CPU seconds for witness validation on the y axis. A caption "p/c" abbreviates "witnesses produced by p that are accepted by c"

Claim 3: Effort and Feasibility 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 be validated quickly, while the validation of another 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 3a 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 3b 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. Figure 3c shows that for Automizer, there is no discernible difference between the CPU times required to produce a witness and to validate it. Figure 3d is similar to Fig. 3a: there are cases for which the validation is faster than the verification and vice versa. Since in this figure, 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.

General Trend. In general, we could not observe a general trend of speed-up over all validation runs. We attribute these results to the fact that it is not trivial to determine which invariants should be exported to the witness, because exporting too much information unnecessarily complicates the validation, while too few or too weak invariants impede the feasibility of the validation. This is further complicated by the fact that an invariant that suffices for one validator may not be sufficient for a different validator. There are, however, individual cases for various different types of verification tasks for which a speed-up exists: CPAchecker, for example, takes about 800 s to verify the task `product-lines/elevator_spec1_product31_true-unreach-call.cil.c`, but validating the witness with CPAchecker takes only about 290 s. It takes CPAchecker 730 s to verify the task `eca-rers2012/Problem15_label35_true-unreach-call.c`, but only about 310 s to validate the witness. Verifying `seq-pthread/cs_peterson_true-unreach-call.i` with CPAchecker takes about 610 s, while validating the corresponding witness with CPAchecker takes about 32 s. As expected, validation only benefits from invariants that are hard to derive but easy to prove. If, on the other hand, too much work is left to the validator, then the validation is slower than the verification. Our prototypical implementations are based on generic model checkers and the potential for optimization towards validation is not yet leveraged.

Placebo Test. To further explore these aspects of witness validation, we performed additional experiments, where we compare the validations of real witnesses with validations of 'empty' witnesses, which contain only invariants *true*: We first took the tasks for which CPAchecker had produced a witness and configured the CPAchecker-based validator to not use any invariants. The results of this benchmark can be used as a baseline for comparing the results for validating the real witnesses. In this experiment, only 1 063 empty witnesses were accepted, while 1 018 were rejected, which is an acceptance rate of 51 %. This is significantly worse than the acceptance rate of 92 % depicted in Table 3, where the invariants were used. Then, we took the tasks where Automizer had produced a witness and again configured the CPAchecker-based validator to not use any of the invariants. In this experiment, only 701 empty witnesses were accepted, while 1 197 were rejected, which is an acceptance rate of 37 %. This is the same acceptance rate as the one for the real witnesses produced by Automizer depicted in Table 3, which matches the observation that the invariants contained in the witnesses produced by Automizer are often too few for the plain k-induction proof without auxiliary invariant generation in CPAchecker's validator to succeed, and also suggests that the apparent speed-up observed in Fig. 3b may just be due to the fact that these are verification tasks that the k-induction proof technique is well-suited for.

Summary. In conclusion, these experiments show that the contents of a witness can make a difference for one of the validators (CPAchecker), while they do not seem to noticeably impact the other validator (Automizer), which in turn is slower than the validator based on CPAchecker. This

trade-off between the reasoning power of a validator and the quality of the witnesses it validates is one of the strengths of our flexible exchange format for correctness witnesses: Users may choose a quick but less powerful validator or a slower but more powerful one, depending on their use case.

5.6 Validity

Benchmark Selection. For our benchmarking, we selected several full categories from the standard repository of software-verification tasks without any selection of subsets. We excluded those categories that are not supported by one of the two verification tools: two memory safety categories, the recursive category, the termination category, and the concurrency category. 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 prototypical implementation is still insufficient. There is a large number of verification tasks for which one of the tools is insufficient, and more engineering effort would be necessary to succeed on those as well.

Verification Tools. Our implementations for producing and validating correctness witnesses are based on two software verifiers that use completely different technologies: in CPAchecker we implemented an approach using k-induction [6] and UltimateAutomizer uses an automata-based approach [17]. This means that comparisons of speed between verification with one tool and validation with the other tool are only meaningful on a very coarse level. For the comparisons between verifier and validator within the same framework, however, no such restriction applies, because the CPAchecker-based verifier and validator differ only on how auxiliary invariants are derived, and the UltimateAutomizer-based verifier and validator are the same except the validator parses and checks the invariants.

Reproducibility. We use the state-of-the-art benchmarking framework BenchExec [10] for measuring and controlling the computing resources (CPU time, memory, core and memory assignment), in order to make sure that our results are accurate and reliable. The data (verification tasks, witnesses, verifiers, and their configurations) are available on our supplementary web site.[6]

6. ALTERNATIVE IMPLEMENTATIONS

The usefulness of an exchange format increases with the tool support for the format. Therefore, it would be good to explore more strategies for producing and validating witnesses. One idea for witness validation that we would like to see implemented in addition to those we already provide is as follows. Take all finite sequences of edges of the CFA of the verification task that unroll the loops at most once. From this set, compute for each finite sequence of transitions

$$\tau := (\ell_i, op_i, \ell_{i+1}) \dots (\ell_{i+n}, op_{i+n}, \ell_{i+n+1})$$

and the witness invariant I_i at ℓ_i the strongest post condition. Then check if the computed strongest post condition $sp(I_i, \tau)$ implies the witness invariant at ℓ_{i+n+1}. If the invariant is not implied, reject the witness. Otherwise, replace the computed strongest post condition by the invariant and continue with the next sequence of transitions. If no finite sequence of transitions is left and the witness was not rejected, accept it.

In the range of potential validation strategies, this technique is an extreme, because it would always require the witness to contain a full proof and would directly reproduce the program abstraction represented by the witness. As with proof-carrying code, the higher this abstraction is, the faster the validation would complete.

7. CONCLUSION

Software verification is a mature research area and there were many breakthroughs in the past two decades that made software verification efficient enough to be applicable on industrial scale. But why is software verification not picked up more in industry? Probably because of usability and trust issues. In testing, an engineer constructs a test suite for a certain coverage and obtains precise results: (i) a quantitative coverage and (ii) a precise answer on which tests passed and which tests failed. Considerable resources are spent, but concrete answers are provided in return. In verification, an engineer has to invest a significant amount of resources, but in turn gets back a wishy-washy answer TRUE or FALSE without any argument. The confidence in this answer is only derived from the reputation of the verification tool. Checking by manual inspection if an error path represents an actual bug or a false alarm is a tedious task and a waste of resources. For the answer TRUE, most tools do not even bother to output any reason why the verifier reports the program as correct.

We propose to change this situation by using machine-readable, tool-independent witnesses for both specification violations and correctness. In this paper we focus on correctness witnesses and suggest a format (a simple extension of the already existing format for violation witnesses [5]) and provide several example implementations for both, producing and validating witnesses. Producing witnesses should be easy, because a proof of correctness is present in every verification tool anyway, if the verifier is designed for more than just bug-hunting. In practice, there are some engineering efforts necessary to compute a useful witness. Witness validation is harder to implement, because the invariants in the witness need to be understood and assigned to the program locations that they were meant for.

We performed a large experimental study with thousands of verification runs on problems from the public repository of verification tasks (C programs). We implemented the approach in two verification tools that performed extremely well in the recent competitions on software verification, and tried to validate witnesses that were produced by the same verification framework and also across verifiers. The results with our proof-of-concept implementations show that the approach can work in practice. We hope that other developers find our ideas useful and implement support for witnesses in their tools, thus adding the value of diversity to the process: While applying a validator to a witness produced by a verifier based on the same components as the validator may serve as a sanity check, a defective common component may hide flaws in the reasoning. Our solution is to establish a common exchange format that many verifiers support, such that different validators that are based on different technologies can be used. So far, there are two validators that are based on two completely different technologies, and our results on witness validation show that this already helps a lot. If witnesses become an accepted standard in software verification, then there will be a lot of tools around that focus not only on witness validation, but also on witness visualization, witness maintenance, quality measures for the invariants or error paths, bug and proof databases, and many more.

8. ARTIFACT DESCRIPTION

This section describes the replication package for our experiments. Our supplementary web page [9] provides all experimental data, and a virtual machine that contains our implementations and that has been prepared such that our results can be easily replicated. In this section, we will give an overview over the proposed exchange format for correctness witnesses, such that the reader may understand our approach and derive an own implementation of our concepts. Further details on how to replicate our experiments inside or outside of the virtual machine can be found on the supplementary web page [9], which also contains a tutorial.

8.1 Witness Exchange Format

Our exchange format for correctness witnesses extends the exchange format for violation witnesses [5] to add the possibility to attach invariants to witness-automata states.

Source-Code Guards. From the existing format for violation witnesses we adopt all source-code guards, such as `startline` and `endline`, which are used to map a transition in the witness automaton to lines in the original program. The source-code guard `control` also continues to be used to distinguish between different branches in the program. Valid values for this guard are `condition-false` and `condition-true`, where for a conditional branching in the original program, the then-branch is referred to by the value `condition-true`, and the else-branch is referred to by the value `condition-false`. Given such a `control` guard, an observer-automaton transition matches if the observed analysis takes the control-flow edge corresponding to the specified branch, but not its counterpart. A source-code guard that we newly introduce for correctness witness, but is also applicable to violation witnesses, is the guard `enterLoopHead`, which signifies that an observer-automaton transition annotated with this guard only matches if the observed analysis takes a control-flow edge into a loop head.

State-Space Guards. In our format for correctness witnesses, we forbid the usage of state-space guards that are used to restrict the state-space exploration for violation witnesses, because the validation of correctness witnesses requires an unrestricted exploration of the state space to ensure that violations cannot be hidden from the validator. Specifically, this ban affects the guard `assumption`, which is currently the only type of state-space guard in witnesses.

States and Invariants. In violation witnesses, automaton states can be annotated with the Boolean data keys `entry`, `sink`, or `violation`, where the default value is `false` if the data key is not present. In contrast, correctness witnesses contain no violation or sink states, because a violation state would contradict the purpose of the witness, and a sink state would restrict the exploration of the state space.

To attach invariants to automaton states, we introduce two new keys for state data tags, namely `invariant` and `invariant.scope`. Valid values for the `invariant` data tag are expressions of the input programming language, such as `(x == y && x > 0)`. All variables used in these invariants must appear in the original program code. Name conflicts between local variables that have the same name as local variables of other functions or global variables can be resolved by using a data tag with the key `invariant.scope` and, as its value, the name of the function that the invariant is intended to be interpreted in. This mechanism is similar to `assumption` and `assumption.scope` in violation witnesses.

[9] https://www.sosy-lab.org/research/correctness-witnesses/

```
44 <graph edgedefault="directed">
45  <data key="witness-type">
   ↪ correctness_witness</data>
46  <data key="sourcecodelang">C</data>
47  <data key="producer">CPAchecker
   ↪ 1.5-svn</data>
48  <data key="programfile">
   ↪ example-safe.c</data>
49  <data key="programhash">
   ↪ 9ea56d387b773690c13645abf30cce9571728660
   ↪ </data>
50  <data key="memorymodel">precise</data>
51  <data key="architecture">32bit</data>
52  <node id="q0"><data
   ↪ key="entry">true</data></node>
53  <node id="q1">
54   <data key="invariant">(y == x)</data>
55   <data key="invariant.scope">main</data>
56  </node>
57  <edge source="q0" target="q1">
58   <data key="enterLoopHead">true</data>
59   <data key="startline">5</data>
60  </edge>
61  <node id="q2"/>
62  <edge source="q1" target="q2">
63   <data key="startline">6</data>
64   <data key="control">condition-true</data>
65  </edge>
66  <node id="q3"/>
67  <edge source="q1" target="q3">
68   <data key="startline">6</data>
69   <data
   ↪ key="control">condition-false</data>
70  </edge>
71  <node id="q4"/>
72  <edge source="q2" target="q4">
73   <data key="startline">7</data>
74  </edge>
75  <edge source="q4" target="q1">
76   <data key="enterLoopHead">true</data>
77   <data key="startline">8</data>
78  </edge>
79 </graph></graphml>
```

Figure 4: Correctness-witness automaton (cf. Fig. 1b) for the introductory safe example program (Fig. 1a) in GraphML format; header omitted for brevity

8.2 Example Witness

Figure 4 shows the witness automaton produced by CPAchecker for the example C program from the introduction section, stripped down to the important parts and without the GraphML header [12]. Line 52 shows the entry state q_0. Lines 53–56 show that (cf. introduction) the witness contains the invariant `(y == x)` at state q_1, using the new data-tag key `invariant`, and that the scope of the variables in this expression is function `main`, using the new data-tag key `invariant.scope`. Lines 61, 66, and 71 declare the states q_2, q_3, and q_4, respectively. Lines 57–60 represent the transition from state q_0 to q_1, marked as entering a loop head using the new source-code guard `enterLoopHead`. Lines 62–65 represent the transition from state q_1 to q_2, corresponding to the then-branch of line 6 in the C program (cf. Fig. 1b, `while` condition fulfilled), while the transition from q_1 to q_3 in lines 67–70 corresponds to the else-branch of line 6 in the C program. Lines 72–78 show the transitions from q_2 over q_4 back to q_1, corresponding to the loop body.

9. REFERENCES

[1] D. Beyer. Status report on software verification. In *Proc. TACAS*, LNCS 8413, pages 373–388. Springer, 2014.

[2] D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Proc. TACAS*, LNCS 9035, pages 401–416. Springer, 2015.

[3] D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses. In *Proc. TACAS*, LNCS 9636, pages 887–904. Springer, 2016.

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

[5] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proc. FSE*, pages 721–733. ACM, 2015.

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

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

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

[9] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[10] D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232, pages 160–178. Springer, 2015.

[11] D. Beyer and P. Wendler. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proc. SPIN*, LNCS 7976, pages 1–17. Springer, 2013.

[12] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report. In *Graph Drawing*, LNCS 2265, pages 501–512. Springer, 2001.

[13] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. PLDI*, pages 66–77. ACM, 2007.

[14] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The Kind 2 Model Checker In *Proc. CAV*, LNCS 9780, pages 502–509. Springer, 2016.

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

[16] M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate Automizer with array interpolation. In *Proc. TACAS*, LNCS 9035, pages 455–457. Springer, 2015.

[17] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *Proc. CAV*, LNCS 8044, pages 36–52. Springer, 2013.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. CAV*, LNCS 2404, pages 526–538. Springer, 2002.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, pages 332–358, 2003.

[20] A. Iliasov. Generation of certifiably correct programs from formal models. In *Proc. WoSoCER*, pages 43–48. IEEE, 2011.

[21] M.-C. Jakobs. Speed up configurable certificate validation by certificate reduction and partitioning. In *Proc. SEFM*, LNCS 9276, pages 159–174. Springer, 2015.

[22] M.-C. Jakobs and H. Wehrheim. Certification for configurable program analysis. In *Proc. SPIN*, pages 30–39. ACM, 2014.

[23] M.-C. Jakobs and H. Wehrheim. Programs from proofs of predicated data-flow analyses. In *Proc. SAC*, pages 1729–1736. ACM, 2015.

[24] T. Kahsai and C. Tinelli. PKind: A parallel k-induction based model checker. In *Proc. PDMC*, EPTCS 72, pages 55–62, 2011.

[25] K. S. Namjoshi. Certifying model checkers. In *Proc. CAV*, LNCS 2102, pages 2–13. Springer, 2001.

[26] G. C. Necula. Proof-carrying code. In *Proc. POPL*, pages 106–119. ACM, 1997.

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

[28] C. Sternagel and R. Thiemann. The certification problem format. In *Proc. UITP*, EPTCS 167, pages 61–72, 2014.

[29] A. Taleghani and J. M. Atlee. Search-carrying code. In *Proc. ASE*, pages 367–376. ACM, 2010.

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

[31] M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In *Proc. FME*, pages 431–450. Springer, 2002.