Witness Validation and Stepwise Testification across Software Verifiers

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

[1] University of Passau, Germany [2] University of Freiburg, Germany

ABSTRACT

It is commonly understood that a verification tool should provide a counterexample to witness a specification violation. Until recently, software verifiers dumped error witnesses in proprietary formats, which are often neither human- nor machine-readable, and an exchange of witnesses between different verifiers was impossible. To close this gap in software-verification technology, we have defined an *exchange format for error witnesses* that is easy to write and read by verification tools (for further processing, e.g., witness validation) and that is easy to convert into visualizations that conveniently let developers inspect an error path. To eliminate manual inspection of false alarms, we develop the notion of *stepwise testification*: in a first step, a verifier finds a problematic program path and, in addition to the verification result FALSE, constructs a witness for this path; in the next step, another verifier re-verifies that the witness indeed violates the specification. This process can have more than two steps, each reducing the state space around the error path, making it easier to validate the witness in a later step. An obvious application for testification is the setting where we have two verifiers: one that is efficient but imprecise and another one that is precise but expensive. We have implemented the technique of error-witness-driven program analysis in two state-of-the-art verification tools, CPACHECKER and ULTIMATE AUTOMIZER, and show by experimental evaluation that the approach is applicable to a large set of verification tasks.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Theory, Verification

Keywords: Error Witness, Counterexample Validation, Software Verification, Program Analysis, Model Checking

1. INTRODUCTION

Software verification becomes more and more important in practice; several breakthroughs in verification research were achieved during the last decade, and several successful verification tools were developed. The TACAS International Competition on Software Verification (SV-COMP) [1] [4, 5] serves as a showcase of the state-of-the-art. Users can choose from a wide range of verifiers, and the SV-COMP categories give an approximate guidance on which verifier is good for which kind of programs. One important and unsolved problem of applying verification technology in practice is that verification tools sometimes produce false alarms, and it still requires an enormous manual effort to find out if a reported bug indeed represents a genuine specification violation.

Our solution comprises two components: we developed an *exchange format for error witnesses* and evaluated its effectiveness by a thorough experimental evaluation, and we develop the notion of *stepwise testification*, as the technique of witness validation immediately leads to the notion of witness refinement, enabling a chain of verifiers (or testifiers) to continuously refine the erroneous state space until a test vector for the error is found.

Testification is the process of giving evidence for a claim that a given program satisfies, or violates, its specification. The evidence of the absence, or presence, of a specification violation is given by one or more witnesses. A verification tool is a *testifier* if it provides evidence to support its claim, i.e., if it produces a witness for correctness or for a violation of the specification. *Stepwise testification* is the process of applying testification in several steps, on ever refined witnesses, possibly using different verification tools, combining different strengths. Figure 1 illustrates the process of stepwise testification. In this paper, we focus on stepwise testification of specification violations by producing error witnesses (left part), while conditional model checking [10] focuses on stepwise testification of correctness.

We accompany the bug report of verifier V_1 with an error witness, which represents information that can effectively guide another verifier V_2 to efficiently re-explore the state space that verifier V_1 reported to contain a bug. Our experimental study confirms the following insights: (1) our exchange format makes it possible to communicate error witnesses across verifiers, (2) verifier V_2 needs on average considerably less resources to validate the witness than verifier V_1 needed to find the error, even if V_2 uses a more expensive verification technology (e.g., V_1 using linear and V_2 using bit-precise arithmetic), (3) stepwise testification can be more efficient than verification, i.e., the CPU time for V_1-verification + V_2-witness-validation can be less than the CPU time for V_2-verification alone, (4) the state-space to be analyzed by V_2 is effectively reduced.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786867

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

Our technique was already used in the most recent edition of the competition on software verification. The SV-COMP community manifested in the competition rules that each answer FALSE must be accompanied by an error witness. Previously, only the *existence* of an error witness was checked, but not its *meaning*. The last edition of the competition rules [5] required the organizer to reasonably validate each witness in order to get more confidence that the error witness indeed represents a valid bug before assigning a success score.

On the syntactic level, we use XML, more specifically GraphML [19], as a language to represent error witnesses. On the semantic level, we use the standard concept of (non-deterministic) finite automata to represent an error witness. A witness automaton observes the paths that the verifier explores and directs the exploration engine along the paths that the witness describes, i.e., towards the violation of the specification. Witnesses can be read by humans (perhaps using a visualization or inspection tool) or a witness validator.

Witness automata allow different levels of abstraction. A most abstract error-witness automaton allows all paths through the program, i.e., it does not restrict the state-space exploration. A most concrete error-witness automaton represents one concrete error path, which is annotated with value assignments for all variables (i.e., a concrete test vector is represented). Many interesting witnesses occur in between the two extremes: there is a continuous spectrum of refined witness automata, reducing the number of possible choices through a program until the most concrete level —the test vector— is reached. Sometimes it is efficient to represent several (or even infinitely many) error paths in one witness.

Contributions. We make the following novel contributions:

- We develop a syntactical *exchange format* based on XML for exchanging and archiving witnesses across software verifiers, and instantiate the format for C programs.
- We introduce *witness automata* as a semantical basis from which a program analysis can be constructed as an extension for existing verifiers (two examples given).
- Our approach validates witnesses against the given program and specification, independently from the computing resource and verifier used to produce the witness, i.e., neither this resource nor this verifier need to be trusted.
- We develop the notion of *stepwise testification*, which continuously refines witnesses using different verifiers or different approaches on perhaps different platforms.
- We provide an extensive experimental evaluation that shows the potential and feasibility of using error witnesses and stepwise testification.

Advantages and Applications. Our solution has various application scenarios in research and industry, for example:

- Witness validation can be used to automatically eliminate false alarms in program analysis (e.g., as used in [30]).
- A common exchange format can boost the development of various tools for collecting, correlating, explaining, and visualizing error witnesses from different sources.
- Error witnesses can accompany bug reports.
- It is sound to use untrusted computing resources and verification engines if followed by witness validation on trusted computing resources and trusted witness validators. This is especially interesting for verification in the cloud.
- Different verification approaches and tools have different strengths, and thus, it is interesting to combine different verification approaches via stepwise testification.

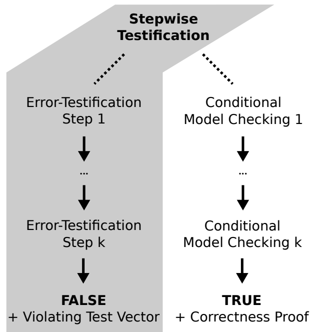

Stepwise Testification
Error-Testification Step 1
Conditional Model Checking 1
Error-Testification Step k
Conditional Model Checking k
FALSE + Violating Test Vector
TRUE + Correctness Proof

Figure 1: Stepwise testification: conceptual view

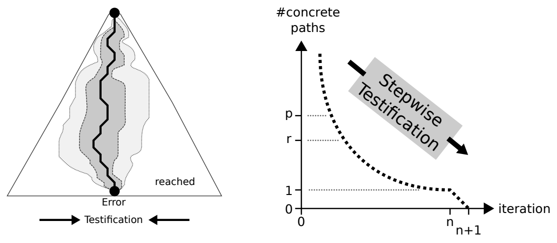


Figure 2: Testification reduces the state space (left) and the number of paths (right) that need to be (re-)verified.

- Error witnesses can describe program parts that should be covered by a (regression) test suite.
- Witness validation implies a notion of witness quality: a more concrete witness (less resources needed for validation) might be considered better than an abstract witness.

Figure 2 (left) illustrates how testification gives evidence of the presence of a specification violation by reducing the state space that the witness validator needs to verify, i.e., the state-space to be explored gets narrower. Figure 2 (right) illustrates that testification reduces the number of paths that the witness validator has to verify. For example, given a program that contains r error paths, an error witness might represent more paths, e.g., p paths, including some infeasible error paths, but testification might further refine such an error witness until only one concrete path is represented (e.g., in iteration n). Testification might also find out that all paths of the witness are not violating the specification (error witness rejected, cf. iteration $n + 1$).

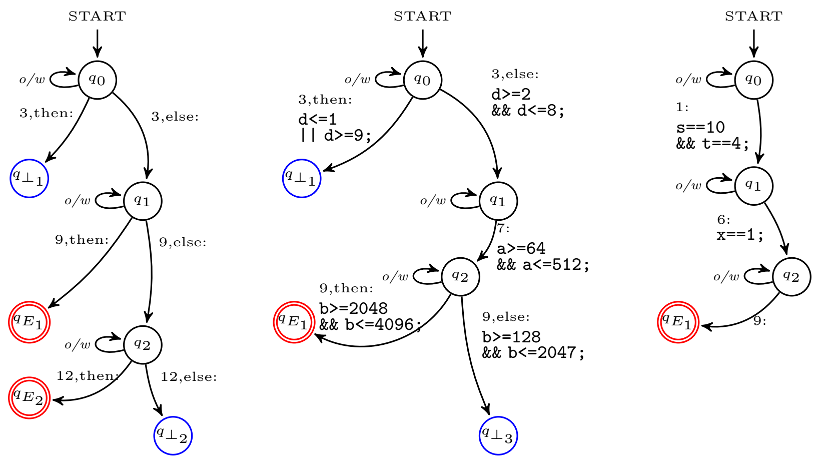
Example. We illustrate how error witnesses are validated and refined across verifiers: We start with an overview of the process and then describe the process of producing and consuming error witnesses in more detail.

We run three verifier instances in sequence. Each of them takes the program shown in Fig. 3a as input, and produces an error witness. The specification of the program is that neither label `ERROR1` nor `ERROR2` is reachable from the program entry. The first verifier runs an analysis based on predicate abstraction [8, 31] with counterexample-guided abstraction refinement (CEGAR) [21], and produces the error witness in Fig. 3b. The second verifier takes this witness (and the program) as input and runs an analysis based on an interval domain [27], and produces the error witness in Fig. 3c. In the last step, we run a test-case generator [6, 13] that takes the error witness from Fig. 3c as input and produces the test vector in Fig. 3d (Witness 3).

Before we continue with the example, we will informally introduce several basic concepts. We model the program as a control-flow automaton (CFA); its locations represent

```
1  int foo(int s, int t) {
2    int d = s-t;
3    if (d < 2 || d > 8) {
4      return 0;
5    }
6    int x = nondet_int();
7    int a = x ? 512 : 64;
8    int b = a*d;
9    if (b >= 2048) {
10     ERROR1: exit(2);
11   }
12   if (b < 128) {
13     ERROR2: exit(3);
14   }
15   while (a > 0) {
16     a--;
17     // code omitted
18   }
19   return b;
20 }
```

(a) Source code



(b) Witness 1 **(c)** Witness 2 **(d)** Witness 3

Figure 3: Example C program (a) and a testification sequence of three witness automata: accepting control states are drawn as double circles, sink states are labeled with the subscript $\perp$, non-accepting control states are drawn as single circle.

program-counter values (we denote the location before the operation on line i as l_i), and its edges represent program operations. Error witnesses are represented as finite automata that describe a set of program paths that should contain at least one path that violates the specification. Paths that end in a sink control state (not accepted by the witness automaton) should not be explored by a verifier that consumes the witness (during witness validation). In our figures, each transition of a witness automaton has a label, which is divided into two parts (by colon): The first part is the *source-code guard*, which is used to *match* the control-flow and can be given, e.g., as a line number. The second, optional part is the *state-space guard*, given as a C expression, which is used to *restrict* the abstract successor state of an analysis, i.e., the set of concrete program states that it represents. The exploration of the state space can be restricted either by transitions to sink control states (that have no outgoing transitions, and thus the path exploration ends) or by restricting the state space using state-space guards at transitions. A state-space guard needs to be checked if the corresponding source-code guard matches the current control-flow edge. We only consider analyses that construct reachability graphs of abstract states. In our example, an abstract state is a tuple (q, l, e) that is composed from the current state q of the witness automaton, the current location l of the CFA, and a component e that represents the information that is tracked by the chosen abstract domain (for example, the interval domain, or the predicate-abstraction domain). The analysis that consumes a witness automaton proceeds along those CFA edges that match transitions of the witness automaton.

The first witness automaton (Fig. 3b) is produced by a verifier with predicate abstraction [8, 31], CEGAR [21], and lazy abstraction [9, 36]. The analysis detects that taking the `then`-branch in line 3 cannot lead to a violation of the specification, and thus, writes a transition to the sink state $q_{\perp_1}$ (with source-code guard `3,then:`), and similarly for the *else* branch in line 12. The multiplication in line 8 is overapproximated by the analysis (this predicate analysis uses linear arithmetics), and thus, it reports paths to both error locations. Despite the infeasible error path that is included in the witness, the loop of lines $15 - 18$ has already been analyzed, i.e., a successive witness validation can ignore the loop completely.

In the next testification step, the error witness in Fig. 3b is validated by a verifier that runs an interval analysis. The verifier starts with the abstract state $(q_0, l_1, \cdot)$, i.e., the witness automaton is in its initial state q_0, and the CFA is on its initial location l_1 (which corresponds to the function entry). The first analyzed operation is the declaration of the function parameters and their initialization. In the witness automaton, only the self transition on state q_0 labeled with "*o/w*" (otherwise) matches; self-transitions such as this one match only if no other transition matches. Thus, the verifier proceeds to line 2 with the abstract state $(q_0, l_2, \cdot)$. The assignment `d=s-t` is matched (again) by the self-transition and the witness automaton stays in q_0. From abstract state $(q_0, l_3, \cdot)$, we compute one successor state for each case of the condition in line 3: one for the `then`-case, and one for the `else`-case; each case is matched by a corresponding edge of the witness automaton. The `then`-branch of the `if`-statement in line 3 is matched by the transition to the sink state $q_{\perp_1}$; the analysis does not continue on that branch because $q_{\perp_1}$ has no successor. Thus, the `else`-branch of the `if`-statement is taken by proceeding to q_1 in the witness automaton. For the assignments of lines 6 and 7, the witness automaton takes the self-transition in q_1, where the analysis has to branch due to the conditional assignment in line 7: on one branch, the value of `a` is 64, on the other branch it is 512. The paths immediately join again and the value of `a` is in the interval $[64, 512]$. The automaton stays in q_1 and the analysis continues (with `b` in $[128, 4096]$ after line 8) until it reaches line 9 with the abstract state $(q_1, l_9, \cdot)$. We compute a separate abstract successor state for each case of the condition in line 9: the successor state for the `then`-case is $(q_{E_1}, l_{10}, \cdot)$, because the next operation (label) `ERROR1` is reached. Since this is the (accepting) error state q_{E_1}, we have confirmed the first error path of the witness. The analysis continues at line 12 by computing successors for (q_2, l_{12}, o_k) which was the result of the `else`-case of the condition in line 9. The state component o_k, which represents the information tracked by the interval analysis, stores the intervals $\{a \mapsto [64, 512], b \mapsto [128, 2047], d \mapsto [2, 8]\}$. The analysis can not take the `then`-branch from q_2 (reaching the accepting state q_{E_2}, which corresponds to `ERROR2` in the CFA), because the interval analysis stored that the value of `b` is in the interval $[128, 2047]$. Therefore, the label `ERROR2` is not reachable (the previous analysis that produced this witness automaton could not detect this because the predicate analysis overapproximated the multiplication in line 8). Proceeding the `else`-branch from line 12, we enter $q_{\perp_2}$ of the witness automaton and reach line 15. The analysis that produced the witness was able to prove that the code from line 15 onwards does not violate the specification. Therefore, the sink control state $q_{\perp_2}$ concludes the analysis.

Upon completion, the interval analysis produces the witness automaton in Fig. 3c. The automaton is constructed by encoding all abstract paths that lead to the violating abstract state in line 10, and adding transitions to sink control states along the paths such that a witness validator can stop the exploration of branches that are not relevant for reaching the violating abstract state.

In the last testification step, the error witness from Fig. 3c is used to restrict a test-case generator [6] in order to derive a specific test vector for the path to `ERROR1`. The test vector is derived by extracting a satisfying assignment of the formula that represents the program path to the error location. The third witness automaton (Fig. 3d) represents the result of the test-case generation, i.e., a test vector.

Related Work. There is a large body of work on combining different verification approaches, starting in the 1970's with *reduced products* of abstract domains [28]. There are many combinations of model checking with data-flow analysis, with testing, and with deduction (cf. [22] for an overview). Most existing approaches combine different techniques in parallel as a product or as portfolio verification. More recently, stepwise sequential combinations were conceptualized as conditional model checking [10], where verification artifacts ('conditions') are passed from one verifier to another. Our contribution focuses on stepwise testification of specification violations, based on passing error witnesses from one verifier to another.

Model-checking tools usually produce and provide counterexamples in some form. However, there is an increasing awareness for the need of (fast and automatic) validation of program error paths in order to increase the confidence in the automatic bug reports, especially to reduce the number of false alarms [5, 18, 29, 40]. A cheap data-flow analysis can be combined with a high-precision feasibility analysis to filter out as many false alarms as possible within one instance of a verifier [29]. Experiments illustrate that the validation of error witnesses is usually significantly faster than re-verifying the (full) program [18]. The tool that produced an error witness can work with a different abstract domain than the tool that is used for the validation [40]. There was, however, no unified exchange format for error witnesses across verifiers.

The process of refining witnesses has been addressed for safety proofs from software verification [10, 26], but not yet for counterexamples. Clarke and Veith [24] mention the fact that already a full Kripke structure with a violation of the specification would be a valid counterexample; they define a set of criteria that make counterexamples more useful for tools and potential users. We extended this idea to stepwise testification. Witness testification can be seen as the counterpart of conditional model checking [10].

The exchange of verification (and inference) results across verifiers has not been discussed in detail before. Counterexamples were exported by deriving full programs from counterexamples in order to provide them as input to another tool [11, 14, 41]. This approach is not feasible for witness validation because the witness needs to be checked against the original program. There is a number of trace formats in the context of distributed high-performance computing, e.g., the Open Trace Format [37], or the MPI trace format [1]; their primary intent is to keep record of events in the system, for example, the exchange of messages between processes. These formats have a strong focus on time-stamped events in distributed systems and are not applicable to our problem. The Certification Problem Format [2] was designed for first-order term-rewrite systems and termination analysis.

Error witnesses have many applications [25, 32, 33, 39, 44], and a common exchange format for verification tools will foster further research in this direction, in particular, on combinations of verification and debugging techniques.

2. PROTOCOL ANALYSIS

Preliminaries. We adopt the same notion of programs to describe the theoretical aspects of our ideas as in previous work [12]. The presentation is restricted to a simple imperative programming language that contains only assume operations and assignments, and all program variables are integers. [3] Programs are represented by control-flow automata (CFA). A *control-flow automaton* $C = (L, l_0, G)$ consists of a set L of program locations, modeling the program counter, the initial program location l_0, which models the program entry, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, each of which models the operation that is executed during the flow of control from one program location to another. All variables that occur in operations from Ops are contained in the set X of program variables.

A *program path* is a sequence $l_0 \xrightarrow{op_1} \ldots \xrightarrow{op_n} l_n$ with $(l_{i-1}, op_i, l_i) \in G$ for $1 \leq i \leq n$ and l_0 is the initial program location. A program path is called *feasible* if there exists a test vector [6] for which the program path can be executed, otherwise the program path is called *infeasible*; a feasible path with concrete test values is called *concrete path*.

Protocol Automata. A protocol automaton $A = (Q, \Sigma, \delta, q_0, F)$ for a CFA $C = (L, l_0, G)$ is a non-deterministic finite automaton with the following components: the finite set Q represents the control states of the automaton, the alphabet $\Sigma \subseteq 2^G \times \Phi$ consists of pairs of a finite set of CFA edges from G and a state condition, the transition relation $\delta \subseteq Q \times \Sigma \times Q$ defines the control-state transitions, the initial control state $q_0 \in Q$ defines the start of the automaton, and the set F contains the accepting control states. We write $q \xrightarrow{\sigma} q'$ if $(q, \sigma, q') \in \delta$ and $q \rightarrow q'$ if there exists a σ with $q \xrightarrow{\sigma} q'$.

Protocol Analysis. A *protocol analysis* for a protocol automaton A is a configurable program analysis (CPA) [12] $\mathbb{O} = (D_{\mathbb{O}}, \rightsquigarrow_{\mathbb{O}}, \mathsf{merge}_{\mathbb{O}}, \mathsf{stop}_{\mathbb{O}})$, which tracks the control state

[2] `http://cl-informatik.uibk.ac.at/software/cpf/`
[3] Our implementations are based on CPAchecker [14] and Ultimate Automizer [34], both of which support C programs.

of a protocol automaton $A = (Q, \Sigma, \delta, q_0, F)$ and consists of the following components (for a given CFA (L, l_0, G)):

1. The abstract domain $D_{\mathbb{O}} = (\mathcal{C}, \mathcal{Q}, [\![\cdot]\!])$ consists of the set $\mathcal{C}$ of concrete states, the semi-lattice $\mathcal{Q}$, and a concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{Q} = (Z, \sqsubseteq, \sqcup, \top_{\mathcal{Q}})$, with $Z = (Q \cup \{\top\}) \times \Phi$, consists of the set Z of abstract data states, which are pairs of a control state from Q (or special lattice element $\top$) and a condition from Φ, a partial order $\sqsubseteq$, the join operator $\sqcup$, and the top element $\top_{\mathcal{Q}}$. The partial order $\sqsubseteq$ is defined such that $(q, \psi) \sqsubseteq (q', \psi')$ if ($q' = \top$ or $q = q'$) and $\psi \Rightarrow \psi'$, the join $\sqcup$ is the least upper bound of two abstract data states, and the top element $\top_{\mathcal{Q}} = (\top, \mathit{true})$ is the least upper bound of the set of all abstract data states. The concretization function $[\![\cdot]\!] : Z \rightarrow 2^C$ is a mapping that assigns to each abstract data state (q, ψ) the set $[\![\psi]\!]$ of concrete states.

2. The transfer relation $\rightsquigarrow_{\mathbb{O}}$ has the transfer $(q, \cdot) \overset{g}{\rightsquigarrow}_{\mathbb{O}} (q', \psi')$ if the protocol automaton A has a transition $q \xrightarrow{\sigma} q'$ such that $\sigma = (D, \psi')$ and $g \in D$. The condition ψ' of the control-state transition is stored in the successor in order to enable a composite strengthening operator [12] to strengthen the successor abstract data state of another component analysis in the composite analysis using information from condition ψ'.

3. The merge operator combines elements with the same control state:

$$\mathsf{merge}_{\mathbb{O}}((q, \psi), (q', \psi')) = \begin{cases} (q', \psi \vee \psi') & \text{if } q = q' \\ (q', \psi') & \text{otherwise .} \end{cases}$$

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

Composition. The protocol CPA is used as one component in a composite CPA, in which other component CPAs track the data and control state, i.e., information about the values of the variables and the control-flow location. Let $\mathbb{P}$ be a component CPA that tracks the abstract data states. In the composite abstract transfer relation, the abstract data states from the protocol CPA $\mathbb{O}$ and the other CPA $\mathbb{P}$ can be used to strengthen the composite abstract successor in the following way: the composite abstract transfer restricts the abstract data state in $\mathbb{P}$ to represent only those concrete states that satisfy the state condition ψ in $\mathbb{O}$. This way, each abstract data state on a program path that the composite CPA explores, always implies the corresponding state condition of the protocol automaton.

Simulation. A *run* of a protocol automaton A for a program path $l_0 \xrightarrow{op_1} \ldots \xrightarrow{op_n} l_n$ is a simulation sequence $q_0 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_n} q_n$ such that every step $l_i \xrightarrow{op_i} l_{i+1}$ is matched by a step $q_i \xrightarrow{\sigma_i} q_{i+1}$ with $\sigma_i = (D, \psi)$ and $(l_i, op_i, l_{i+1}) \in D$ and all variable assignments at program location l_{i+1} satisfy ψ. Protocol automaton A *accepts* the run if $q_n \in F$. We say A accepts the program path π if there exists an accepting run of A for π. The projection of an accepted run to its alphabet symbols $\sigma_1 \ldots \sigma_n$ is called *accepted word.* The set of all accepted words of A defines the *language* $\mathcal{L}(A)$.

Specification by Observer Automata. Safety properties and security aspects [42] can be modeled using finite observer automata (also called 'monitor automata') that run in parallel to the system to be verified and 'observe' the behavior of the system without influencing it. Observer automata are an established concept for providing a formal specification [3, 7, 14, 43]. Separating specification from implementation supports the idea of separation of concerns; tools can support the user in providing and maintaining the specification. A software specification is either weaved into the source code before verification (cf. SLIC [3] and BLAST [7]), or checked on-the-fly in parallel to the program (cf. BLAST-CPA [43], CPACHECKER [14], ORION [29]). A set of properties can be checked simultaneously within one run of a verifier.

An *observer automaton* is a protocol automaton that satisfies the following condition: for every control state q of A, and every control-flow edge g of C, the state conditions can partition, but not restrict, the state space of the program: $\bigvee \{\psi \mid \exists q' \in Q : \exists \sigma \in \Sigma : \exists D \subseteq G : q \xrightarrow{\sigma} q', \sigma = (D, \psi), g \in D\} = \mathit{true}$.

An *observer analysis* is a protocol CPA for an observer automaton, i.e., an observer analysis 'observes' (or 'monitors') the paths of the observed program, but does not restrict the exploration of the program analysis. The observer CPA can be used to split abstract paths to observe them separately.

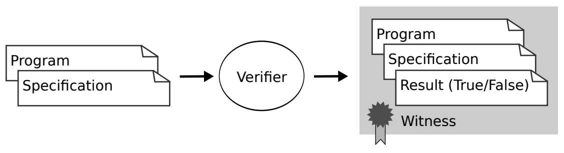
3. TESTIFICATION

The goal of our work is to represent error paths — paths through the program source code that violate the specification— in such a way that they are reproducible, machine-readable, and exchangeable between various verification tools. Conceptually, a witness is information that provides evidence of the verification result. This paper focuses on error witnesses, i.e., witnesses that provide evidence that the given program violates a given specification. For the representation of error witnesses we use witness automata.

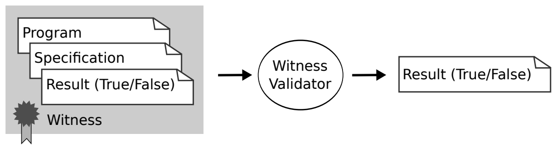
Witness Automata. A *witness automaton* is a protocol automaton, and a *witness analysis* is a protocol CPA for a witness automaton, which runs as one component CPA of a composite program analysis in parallel to other component CPAs. One of the other component CPAs is an observer CPA that encodes the specification. In contrast to observer automata, witness automata not only observe, but also *restrict* how the program analysis explores the program's state space. While an observer automaton has abstract successor states for *all* concrete successor states, a witness automaton can restrict the successor states to those successor states the lead the exploration towards the specification violation. In other words, the witness automaton guides the program analysis to explore the state space that violates the specification. Automata that guide the analysis towards specific program locations are also used for test-case generation [13].

Verification with Witnesses (Testification). We require a verifier, whenever a violating program path is found, to produce a witness automaton for exemplifying the violation; we call such a verifier also *testifier*, because it testifies that the bug exists, and we call this process *testification*, as illustrated in Fig. 4a. The purpose of the witness automaton is to later restrict the state-space exploration of a verifier such that the error path can be confirmed with less effort than with a completely independent verification run. In this paper, we focus on *error* witnesses only.

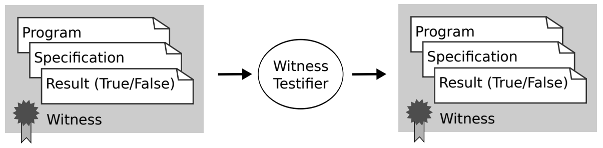
Witness Validation. *Witness validation* is the process of determining if, for a given program, specification, result, and witness, the same result can be re-established independently (Fig. 4b). In this paper specifically for error witnesses, we validate if an error path can be found in the state space that the witness describes. One way of implementing witness validation is to construct a composite program analysis that has



(a) Verification with witnesses (Testification)



(b) Validation of witnesses



(c) Stepwise testification with witnesses

Figure 4: From verification and witness validation to stepwise testification

a specification analysis and a witness analysis as components, which simultaneously observe and restrict the state-space exploration: the witness validation restricts the search of the composite program analysis such that only paths are explored that the witness automaton can match, and the specification analysis checks if the path indeed violates the specification. If, during the analysis of a program path, the witness automaton takes a transition to a sink state, the analysis stops exploring the path, thus, restricting the state-space exploration. The witness is confirmed by the witness validator if both, the specification automaton and the witness automaton, take a transition to their respective (accepting) error control state. We call such a verifier *witness validator*.

A widely-used instance of error-witness validation is counterexample checking (e.g., [30]), for example during the refinement phase in CEGAR [21], where an abstract counterexample (which is an error witness) is checked for feasibility (the witness testifies against the program's claim to correctness). The given error witness is either rejected, which means that it describes no feasible counterexample, or it is accepted because it contains a concrete error path and CEGAR stops.

Abstraction Levels of Witness Automata. A witness automaton can represent more than one error path; in fact, the verifier that constructs the witness automaton is not bound to a certain level of abstraction. Obviously, for the purpose of witnessing a violation of a specification, the witness is the better the more the witness automaton restricts the search space, in order for the validating program analysis to explore fewer paths and validate the witness faster.

Stepwise Testification with Witnesses. Stepwise testification of witnesses is the iterative process of improving witnesses, by removing unnecessary state space. This process combines witness validation with testification: it takes as input a witness and produces as output a better witness, as illustrated in Fig. 4c. Here, for error witnesses, a testification step starting with witness w produces a witness w' that describes a subset of paths that contains an error path. Each testification step reduces the state space that the next testification step has to explore.

The most concrete level of error witnesses describes one single concrete path that violates the specification (cf. Fig. 2, right). Such a concrete error path contains value assignments for all variables, and is equivalent to a test vector.

Stepwise Testification Across Verifiers. In order to enable witness validation and stepwise testification across different verifiers, we propose a verifier-independent *exchange format*, which was already successfully used in SV-COMP 2015 [5]. The exchange of error witnesses across different verifiers enables a wide range of applications, some of which were outlined in the introduction. The following section gives more detail on the proposed exchange format.

4. EXCHANGE FORMAT

Using a common exchange format for (error) witnesses, tools that support stepwise testification may be chained arbitrarily. We instantiate these concepts for programs written in C, and an exchange format based on XML.

4.1 Witness Exchange Format

Our format for exchanging error witnesses is based on GraphML [19], an XML-based format for storing and exchanging graph structures. A number of libraries support reading and writing GraphML (or at least XML), and thus make the adoption of the format convenient. The graph nodes and graph edges represent the control states and the transitions of the witness automaton, respectively.

We make use of the extensible nature of GraphML to extend it with custom data for storing error-witness information. Both the node and edge elements in GraphML can take additional data within a `data` tag that has an attribute `key`. The meaning of a data tag is determined by its `key` attribute. More details on the error-witness format can be found on our supplementary web page.[4]

Automata States and Transitions. The control states Q and the corresponding transitions $\delta \subseteq Q \times \Sigma \times Q$ of the witness automaton are the central information to be encoded in the format. The format also encodes the different roles that a control state can take: (1) the initial state q_0 of the automaton, (2) a sink state $q_\bot$, (3) an error state q_E, or (4) a 'normal' state. Depending on the role we add a data tag with the key attribute (1) `entry`, (2) `sink`, or (3) `violation`, with the value `true` (the default is `false` for all three, declaring the control state as 'normal').

The behavior of transitions between control states is defined by a set of *guards*. The conditions at the transitions represent source-code guards and state-space guards. A *source-code guard* is used by the validator to check if a control-flow edge of the CFA matches a control-state transition based on the source code. Line numbers are an example of such guards. *State-space guards* are used to strengthen the abstract successor states that are computed by the transfer relation of an analysis after the source-code guard matches the transition to the next control state of the witness automaton. Assumptions like `x == 0; y > 5;` are examples of such guards. A summary of the guards that we use for our experiments is given in Sect. 7.1.

4.2 Format Implementation

This section provides guidance towards robust implementations for writing and reading error witnesses, describes some

[4]`http://sosy-lab.org/~dbeyer/verification-witnesses/`

best practices, and presents how common real-world issues with exchanging (error) witnesses across different verifiers can be addressed.

Writing a Witness Automaton from an ARG. A verification tool can produce witnesses by transforming the desired parts of the abstract reachability graph (ARG) [9] (which is often available from an analysis) into a witness automaton. The nodes along an error path in the ARG become control states in the witness automaton. The initial control state q_0 of the witness automaton corresponds to the root node of the ARG. For every ARG node that violates the specification, we add a corresponding accepting (error) control state. The edges of the ARG become transitions in the witness automaton. Edges that leave the error path in the ARG become transitions to a sink state, i.e., a state with no outgoing transitions. Based on the desired level of abstraction, we add source-code guards to the transitions. Those can be based, for example, on the line number, character offset, and branching (control case) information of the ARG edge. Constraints on variable values at the target state of an ARG edge may be encoded as state-space guards. After producing the witness automaton from the ARG, we perform several minimizations: for example, we remove transitions without any guard, and reduce sequences of similar transitions, i.e., given a sequence of transitions with the same set of guards, we often leave only one of those transitions in the automaton.

Reading a Witness Automaton for Validation. An error-witness validator reconstructs the witness automaton that is stored in our exchange format. For every node in the witness file, there is a state in the witness automaton. For every edge in the witness file, there is a transition in the witness automaton, guarded by, for example, information about line numbers, branchings, and assumptions from the data tags. We add an unguarded self-transition to each accepting (error) state in order to express that given a program path t, for any prefix t' of t that causes the automaton to switch to an error state, t is also an accepted error path. To states that are neither sink nor accepting (error) state, we also add a self-transition that is guarded by the negation of the disjunction of the source-code guards of all other outgoing transitions of this control state, such that it matches if no other outgoing transition matches. We label these transitions with "*o/w*". This is done in order to address differences in program representations between verifiers, which we will discuss in more detail below. After the witness validator has reconstructed the witness automaton, it is able to apply the concepts described in Sect. 3 to validate the witness.

Addressing Differences in Program Representations Between Verifiers by Stuttering. The ANSI C standard specifies that it is not required to evaluate all parts of an expression if some parts are sufficient to determine its outcome. This has to be considered during witness validation. Therefore, in addition to the transitions that are explicitly defined in the error witness, control states that are neither sink control state nor accepting control state have the transition "*o/w*" to itself. This self-transition is required in all cases where the verifier that produced the witness skipped parts of the code, either due to simplifications or optimizations in the front-end of the verifier, or simply due to a higher level of abstraction than applied by the witness validator. For instance, the witness automaton might not contain any information about a block of C code that is unreachable. We consider imposing rules about what to include in witnesses and what can safely be omitted to be too restrictive and a barrier to applying front-end optimizations. Therefore, the witness automaton is allowed to stutter until the actual analysis catches up in its exploration. On the other hand, this implies that a witness validator must use an internal representation of the program that is at least as fine-grained and complete as the witnesses. Another effect of this solution is that the end of a path must be explicitly marked as a sink control state by the witness writer if the state-space guards are not strong enough to contradict the abstract state of the witness validator: otherwise, the state-space reduction may not be as effective, because the witness validator has to consider all remaining branches as well.

Handling Ambiguity. In some cases, the source-code matching information at a transition might be ambiguous. One line might contain several statements:

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

A witness automaton might have the assumption `x==3` after a match of line 2, because it is valid after the third statement in line 2. An analysis that would require the truth of `x==3` after the first statement (first match) of line 2 could terminate the analysis since the path formula $x = 1 \wedge x = 3$ is unsatisfiable. These cases can be identified by considering the direct successor edges in the CFA (lookahead). For a given witness transition with source-code guards, an additional transition from the source control state back to itself is introduced. This transition is guarded by the same source-code guard and additionally by the condition that two successive CFA edges (hence the lookahead) match these guards. If this case is encountered during the analysis, the transitions provide a non-deterministic choice between proceeding to the next control state or staying in the current control state. The downside of this approach is that the non-determinism inflates the state space. However, a smart witness validator may detect cases where there is no semantic difference between the choices. The witness validator then makes a choice, thus avoiding the state-space inflation. For example, if the one-line sequence `int x = 0, n = nondet();` is encountered, it does not matter which of the two declarations an assumption about variable `x` is applied to, because the initialization of `n` does not change the value of `x`. In the given example, the ambiguity can also be resolved by the witness writer by providing an exact offset value.

Imprecise Witnesses. A verifier that runs an analysis with a coarse precision might not be able to choose a specific path to the error location that is feasible in the concrete program. Instead of choosing an arbitrary path, we produce an automaton that describes several paths, assuming that one of these paths is feasible in the concrete program. A witness validator can therefore produce a refined version of the error witness that describes a smaller number of paths.

Best Practices. Different verifiers use different internal representations of programs. An exchange of error witnesses across verifiers is only possible if the witness automata — especially their transitions— refer to program fragments that can be identified by all verifiers in the testification chain. Therefore, we recommend brevity for witness production: Instead of creating several transitions about the same program operation, the verifier should try to merge them into a single transition. Conversely, a witness validator must not prune parts of the source code that might be referenced by a witness. Due to these requirements for witness validators, it is arguably more difficult to implement a witness validator than a verifier that only writes witnesses, but we consider this to be an acceptable trade-off for the flexibility that is gained by enabling a verifier to refine and improve upon the results of other tools.

Open Problems. There are several open problems that we do not yet address in the current error-witness exchange format. Our format does not define a means to specify context switches, so it is currently not possible to express witnesses to specification violations that depend on concurrency. A counterexample to termination, or more general, a counterexample to a liveness property, is not a finite execution. Yet, we can use witness automata to narrow down a program to a set of infinite paths that contains an infinite execution. However, the current syntax does not allow to distinguish loops that may be executed infinitely often and loops that may be executed for an arbitrary but finite number of times.

5. EXPERIMENTAL EVALUATION

To demonstrate the effectiveness and efficiency of error-witness validation, we performed a large number of different experiments. The experimental work flow consists of instructing the verifier (1) to produce an error witness and (2) to validate an error witness.

Experiment Goals. We perform a feasibility study to support the following claims:

Claim 1: We developed an error-witness format that is machine-readable and can be used to exchange witnesses for bugs in C programs between different verifiers.

Claim 2: Witness validation can take considerably less effort than verification, i.e., the witness successfully guides the verifier through a considerably smaller state space.

Claim 3: A high-precision witness validator may improve the overall effectiveness if an efficient but low-precision verifier produces witnesses and the validator rejects a substantial number of incorrect witnesses.

The witness validator is only applied to a verification task (a program and its specification) if the previous verification step produced a witness. An error witness may describe only a subset of the (possibly infinite) paths that violate the specification; rejecting an error witness does therefore not imply that the verification task satisfies the specification, but only that the given error witness does not encode a violating path. Therefore, in our experiments that are restricted to *error* witnesses, the result of witness validation is to be interpreted as follows: FALSE means *witness confirmed*; TRUE means *witness rejected*, i.e., the part of the program that the witness describes does not violate the specification.

Benchmark Set. Our benchmark is composed of the 3 964 verification tasks from all categories of SV-COMP 2015 [5] except *Arrays*, *BitVectors*, *Concurrency*, *Floats*, *MemorySafety*, *Termination*, and *Recursive*, which are not supported by one or both of the evaluated verifiers. A total of 1 148 of these tasks contain known specification violations.

Experimental Setup. All 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 14.04 (64 bit), using Linux 3.13 and OpenJDK 1.7. Each verification task was limited to two CPU cores, a CPU run time of 15 min and a memory usage of 15 GB. CPAchecker was used in revision 17283 from the trunk, with MathSAT5 as SMT solver. CPAchecker was configured to perform a predicate analysis, using the theory of linear arithmetic over integers and uninterpreted functions. Ultimate Automizer was used in revision 14553 from the trunk and used its SV-COMP'15 configuration with Z3 as SMT solver. The benchmarks were executed using BenchExec [17] in version 0.5.

Presentation and Availability. The results, tools, and verification tasks that we used in our evaluation are available on our supplementary web page. [4]All reported times (CPU time) are rounded to two significant digits. For verification runs where the verifier claims to have found a bug, we distinguish between expected alarms, which are alarms for programs with a known specification violation, and unexpected alarms, which are alarms for programs without known specification violations. Unexpected alarms may occur if, due to a lack of precision, a verifier reports an infeasible error path. An expected alarm can still be a false alarm, if the witness is incorrect, i.e., it does not describe a feasible error path. Our knowledge about existing violations is based on the verdicts of the SV-COMP community.[5]

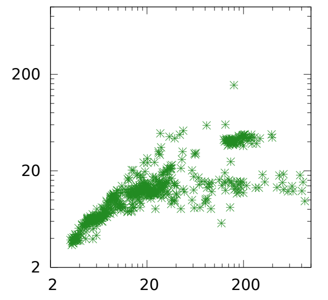
Claim 1: Witness Validation Across Verifiers. Our first experiment represents a feasibility study showing that we were able to implement a witness exchange format for bugs in C programs for two different verifiers, CPAchecker and Ultimate Automizer, which both can take the roles of a verifier (producing witnesses) and a witness validator.

Results. We produced error witnesses with CPAchecker and Automizer, and then used them as input for both verifiers, so that each verifier had to validate its own witnesses, and the witnesses of the other verifier. In total, CPAchecker produced 634 witnesses, 38 of which are considered to be unexpected alarms, and Automizer produced 309 witnesses, 15 of which are considered to be unexpected alarms.

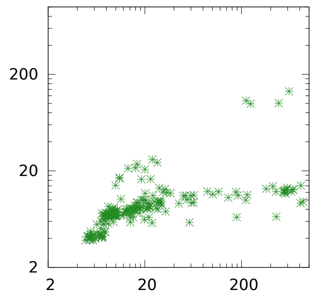
Validation Times. Figure 5a displays the results for validating witnesses produced by CPAchecker with CPAchecker itself and shows that validating the witnesses is at least as fast, and often much faster, than producing the witnesses. This confirms our expectations and matches the results of earlier work [18], where an internal, non-exchangeable witness format was used. The same trend is observable in Fig. 5b, which displays the results for cross-validating witnesses produced by Automizer with CPAchecker, and in Fig. 5c, which displays the results for validating witnesses produced by Automizer with Automizer itself. We also note that Automizer has a slightly higher start-up time (about 6 seconds) than CPAchecker (about 3 seconds) before any results are returned. Figure 5d displays the results for cross-validating witnesses produced by CPAchecker with Automizer. Due to the different start-up times observed above, there are some verification tasks where validating the witnesses with CPAchecker was quicker than validating them with Automizer, but the trend of witness validation often being faster than verification is observable here, too.

Acceptance and Rejection Rates. In order to show that the verifiers actually use and check the witnesses provided to them, a closer inspection of the witness acceptance and rejection rates is required. Table 1 shows the total amounts of accepted and rejected witnesses for each of the four combinations. Runs where the witness validation timed out are counted as rejections. The high rate of accepted expected alarms across verifiers, 69 % and 70 %, respectively, confirms the earlier

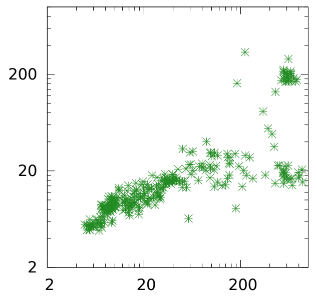
[5]`https://github.com/dbeyer/sv-benchmarks`



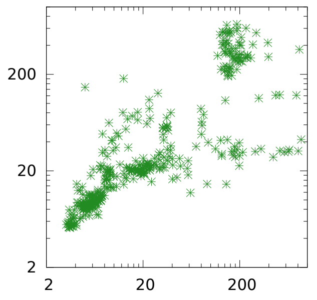
(a) CPAchecker / CPAchecker



(b) Automizer / CPAchecker



(c) Automizer / Automizer



(d) CPAchecker / Automizer

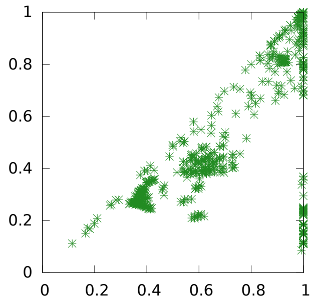
Figure 5: Scatter plots for pairwise composition for witness validation: CPU seconds for producing a witness on the x axis, CPU seconds for witness validation on the y axis. A caption "p/c" abbreviates "witnesses produced by p that are accepted by c"

Table 1: Accepted and Rejected Witnesses

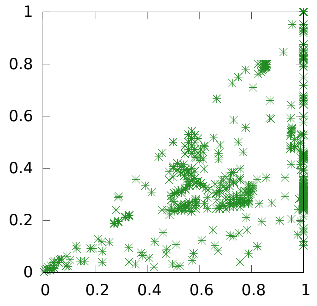
Validator	CPAchecker		Automizer	
Producer	CPAchecker	Automizer	CPAchecker	Automizer
Accepted	615	205	419	305
Rejected	19	104	215	4
Expected alarms only:				
Accepted	585	204	418	291
Rejected	11	90	178	3
Accept. rate	98 %	69 %	70 %	99 %
Unexpected alarms only:				
Accepted	30	1	1	14
Rejected	8	14	37	1
Reject. rate	21 %	93 %	97 %	7 %

observation that the tools understand each other's witnesses. The high rejection rates of 93 % and 97 % for the unexpected alarms across the tools, on the other hand, confirm that our approach is suitable for increasing the confidence in counterexamples. It is also important to notice that not all expected alarms are in fact accompanied by correct witnesses. A verifier may produce an incorrect witness that does not describe a feasible error path while missing the actual bug. For example, CPAchecker produces a witness for the verification task `elevator_spec3_product31_false-unreach-call.cil.c`, but Automizer rejects this witness. Manual inspection reveals that the witness contains a sequence where the value 200 is assumed for a variable `__cil_tmp4` in line 3673 of the verification task, followed by assuming the value 0 for the result of the expression `__cil_tmp4 / 3`, in line 3675 of the verification task, an error caused by a technical limitation of the chosen SMT configuration. The rejection of the described path is therefore justified and desirable. For validating their own witnesses, both the high rate of accepted expected alarms (98 % and 99 %) and the low rate of rejected unexpected alarms (21 % and 7 %) is to be expected, because both the verification run and the witness validation are equally (im)precise, and thus verifiers will repeat the same mistakes they made when producing the witnesses.

Claim 2: State-Space Reduction. In order to support the claim that the guidance provided by witnesses is able to significantly reduce the state space that has to be explored by the validator, we compare the code coverage between producing witnesses and validating those witnesses as an approximate indicator of the explored state space. The term code coverage in this section refers to the code that is visited by the verifier in order to determine the verification result. Another indicator that could be used would be the amount of states in the abstract reachability graphs, however, this measure would not be comparable across different abstract domains, and even though we used the same abstract domain for both configurations in our comparison, we want our results to be comparable to future experiments. For this experiment, we chose the predicate analysis of CPAchecker. We applied two code-coverage measures: line coverage, which counts each visited line once, and condition coverage, which counts each visited case of each boolean sub-expression once.



(a) Line Coverage



(b) Condition Coverage

Figure 6: Scatter plots that illustrate a reduction of the code coverage with values for the verification runs on the x axes and values for the witness-validation runs on the y axes

Results. Figure 6a illustrates that the line coverage of the witness-validation run never exceeds and is often significantly lower than during the initial verification run. Figure 6b shows the same trend for condition coverage, but with an even more extreme reduction of coverage during witness validation in comparison to the initial verification run. Condition coverage is a better indicator for the reduction of the number of explored program paths.

Claim 3: Witnesses Improve Effectiveness and Efficiency. One of our claims is that for many verification tasks, it is faster to use a quick, low-precision verification followed by a high-precision witness validation (thereby obtaining high-precision confidence in the counterexamples), than to use the slow, high-precision verification exclusively. To support this claim, we selected the *DeviceDrivers* benchmark set of SV-COMP'15, which contains 1 650 verification tasks, 203 of which contain a known specification violation. We used two different predicate-analysis configurations of CPAchecker, one using linear arithmetic (`LA`) and one using bit-precise arithmetic (`BP`). We used both configurations to validate the witnesses produced by `LA`.

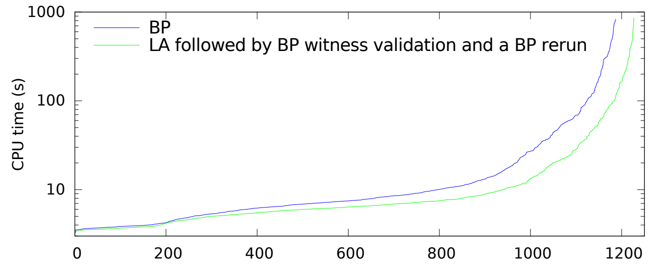


Figure 7: Quantile plot showing that `LA` followed by a `BP` witness validation and rerunning `BP` on rejections is more efficient and effective than just running `BP`

Results. We made the following observations:

1. `BP` produces 1 128 expected proofs, while `LA` produces 1 164 expected proofs. The difference of 36 results is because the bit-precise configuration `BP` times out on tasks for which `LA` is quick enough to provide a proof.
2. `BP` produces 59 expected alarms for programs with known specification violations, and 10 unexpected alarms for programs without a known violation, whereas `LA` produces 77 expected alarms and 14 unexpected alarms (91 in total). The confidence in the alarms reported by `LA` is initially restricted to the limitations of linear arithmetics.
3. 50 of the 91 witnesses produced by `LA` were confirmed by a `BP` validation. The confidence in these remaining witnesses is thus strengthened by the bit-precise analysis. Among the 41 rejected witnesses were 11 of the 14 unexpected alarms. 3 witnesses for tasks where both configurations produced unexpected alarms were confirmed by `BP`.
4. The sum of CPU time required to verify the tasks with confirmed witnesses with `LA` and confirming those witnesses with `BP` amounts to 0.74 h, while verifying the same verification tasks with `BP` takes 3.1 h.
5. If we sum the CPU time of all tasks with `LA`-verification and `BP`-validation and add the sum of CPU time used by `BP` to verify those tasks from scratch for which the `LA`-witness was rejected by `BP` (which is 4.2 h), we obtain a total CPU time of 53 h. This is 9 h less than the total CPU time for verifying all tasks with `BP` (62 h), even though we now have 1 164 expected proofs as well as 66 witnesses in which we have bit-precise confidence, only 3 of which are unexpected alarms (50 from `LA` which were confirmed by `BP` and 16 from consecutively running `BP` on the tasks for which the witness was rejected). This effect is visualized in the quantile plot in Fig. 7. We know that the `BP` rerun could also solve another 9 tasks, however, the sum of the CPU time for running `LA`, validating the witnesses, and rerunning `BP` would exceed the time limit in these cases.

In summary, we observe that for this set of verification tasks, using plain bit-precise predicate analysis for verification is slower than verifying the tasks with predicate analysis with linear arithmetics (`LA`) followed by witness validation with a bit-precise predicate analysis (`BP`), while both provide the same bit-precise confidence in the counterexamples. At the same time, the efficiency of the predicate analysis with linear arithmetics allows to produce more proofs before the timeout than the bit-precise predicate analysis. Therefore, we propose a workflow where the tasks are first verified by `LA`, the witnesses of `LA` are validated by `BP`, and tasks for which the witnesses were rejected are verified by `BP`.

As a sanity check, we also validated the witnesses produced by `LA` with `LA`. As expected, almost all of the witnesses are confirmed, so we are confident that the increased precision of the `BP` witness validation is really the cause of the observed witness rejections, and that the witnesses confirmed by `BP` are valid even with bit-precise semantics, even though they were produced by an analysis restricted to linear arithmetics.

Validity. We chose the *DeviceDrivers* benchmark set for this experiment, because it contains real-world C code, and, contrary to some of the other SV-COMP'15 benchmark subsets, it contains many programs that are large in terms of lines of code and use a wide variety of the programming-language features, such as structs, pointers, pointer arithmetics, and arrays. For verification tasks from other sets that are not as complex, we observed that the bit precise analysis often performs as well as the predicate analysis using linear arithmetics. In our experiment, the goal was to show that witnesses can be used to improve the results for cases where the more precise analysis is too slow, so it would be invalid to include tasks where this premise does not hold.

6. CONCLUSION

If the goal is as ambitious as verifying large software systems, it is required to combine the strengths of different verification techniques and as a precondition, unify their results and make them exchangeable. The objective of our work was to close the gap of a missing exchange format for error witnesses and to establish witness validation as part of the verification and validation process. It is important to have such a format in the verification community, in order to eliminate false alarms by witness validation, and in order to leverage the potential of combining different verification tools: Once a verifier outputs its error witness in the exchangeable format, no extra implementation work is necessary for witness validation; the witness can be given directly as input to an off-the-shelf external witness validator (we provide two open-source implementations). Our experience shows that it is feasible to support such a format, despite the differences in the internal representation of the control-flow of programs.

Stepwise testification goes one step beyond witness validation: Thanks to exchangeable error witnesses, it is easy to design verification and validation processes where various tools can interact, complementing and enriching each other. Our experiments reveal that it is worthwhile to chain testifiers that produce increasingly more restricted witnesses.

We performed an extensive experimental evaluation to show that (1) the proposed error-witness exchange format is flexible enough to enable the combination of two very different verification tools, (2) witnesses effectively reduce the state space that a validator needs to explore, and (3) witness validation is on average significantly faster than verification without the guidance of a witness, and witness validation can confirm and accept witnesses with reasonably good acceptance and rejection rates. Considering that this is the initial study on witness validation, the results look very promising to continue in this direction. Evidence of the success of witness validation was also given by its application in the last edition of the competition SV-COMP 2015.

Acknowledgments. We thank Marek Trtík and Michael Tautschnig for the discussions on a preliminary version of the witness format and the participants of SV-COMP 2015 for supporting error witnesses and providing valuable feedback. We thank the reviewers for their comments to improve the experimental evidence and presentation of our contributions.

7. APPENDIX: REPLICATION PACKAGE

This paper comes with a replication package, which has been successfully evaluated by the *Replication Packages Evaluation Committee.* Our supplementary web page [4] provides all experimental data, and a virtual machine that contains our implementations and has been prepared such that our results can be replicated easily. In this section, we will give an overview over our provided reference implementations, 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 our virtual machine can be found on the supplementary web page.[4]

7.1 Implemented Guards

In our experiments, we used several of the source-code and state-space guards that are mentioned in Sect. 4.1. The most important ones are listed below.

Source-Code Guards. The guards `startline` and `endline` are source-code guards that map a transition to lines in the program. Valid values are integer numbers that correspond to actual line numbers in the original program. A transition matches if the observed analysis takes a control-flow edge that starts at the given `startline` and, if applicable, ends at the optionally given `endline`. We strongly recommend `startline` as a baseline to be supported by all implementations, such that verifiers (testifiers) can rely on their witnesses being consumable by a wide range of witness validators. Program line numbers are an established concept in computer science that programmers are familiar with: there is tool support for navigation based on line numbers in development environments, debuggers, and editors. The source-code guard `control` is used to distinguish between different branches in the program. Valid values for this guard are `condition-true` and `condition-false`, where for a conditional branching in the original program, `condition-true` refers to the then-branch and `condition-false` to the else-branch. The transition matches if the observed analysis takes a control-flow edge that represents the specified branch of a branching edge, but not its counterpart.

State-Space Guards. The guard `assumption` is the only state-space guard that we used so far. Valid values for this guard are expressions of the input programming language, such as `x == 0; y > 5;`. The variables used in these assumptions must appear in the original program code (auxiliary variables that verifiers use internally are not allowed to appear in these expressions). Local variables that have the same name as global variables or local variables of other functions can be qualified by using a tag with the key `assumption.scope` (see below). After the witness automaton takes a transition with an `assumption` guard, the assumptions can be used by the analysis to reduce (strengthen) the state space. By using the syntax of the input programming language for expressing the assumptions, parsing the expressions becomes an easy task for a witness validator; the same parser that was already used to parse the program can be reused. With this decision, we follow the examples of the ANSI/ISO C Specification Language [6] and the BLAST Query Language [7]. The witness validator must map the variables in the given assumptions to the variables in the program. Due to scopes, there may be name conflicts. We propose to explicitly state the variable scope along with the assumption; therefore we use the key `assumption.scope`. Valid values for tags with this key are names of functions that are defined in the program. The witness validator will first look for a variable with a matching name in the scope of the provided function name before checking the global scope. The value of a data tag with this key applies to the assumption as a whole. It is not possible to specify assumptions about local variables of different functions.

[6]`http://frama-c.com/download/acsl_1.8.pdf`

7.2 Witnesses in CPACHECKER

In CPACHECKER, the successor of a witness control state for a given CFA edge is computed by matching the source-code guards of the transitions against the CFA edge as described earlier, such that there is one successor control state for every transition that matches the CFA edge. During the strengthening phase of the configurable program analysis (CPA), the other component program analyses that run in composition with the witness CPA may strengthen their information based on the state-space guards of the transition. This strengthening from witness assumptions is currently implemented for the value analysis and the predicate analysis. For the value analysis, we conjunct the abstract state with concrete variable values (the new values were previously unknown, or contradict the value analysis state and thus make the abstract state unreachable, which results in reducing the state-space that has to be explored). For the predicate analysis, the assumption is conjuncted to the path formula. For more information on these analyses and their implementation within CPACHECKER, we refer to our previous work [15, 16].

7.3 Witnesses in ULTIMATE AUTOMIZER

The validation of witnesses in ULTIMATE AUTOMIZER is done in two steps. In the first step, a new CFA is constructed. The paths of this CFA represent the paths of the original CFA that comply with the source-code guards of the witness automaton: the new CFA is constructed as a product of the original CFA and the witness automaton. The nodes of this product are pairs (l, q), where l is a location of the CFA and q is a control state of the witness automaton. The product contains an edge from (l, q) to (l', q') labeled with op if (l, op, l') is a CFA edge and

1. $q \xrightarrow{\sigma} q'$ is a transition in the witness automaton such that the CFA edge (l, op, l') is one of the edges that is represented by σ,
2. there is an epsilon transition from q to q', or
3. the states q and q' coincide (implicit stuttering edge).

Note that in the current implementation, the state-space guards and the source-code guard `control` of the witness are ignored.

In the second step, ULTIMATE AUTOMIZER verifies if the resulting CFA satisfies the specification using an automata-theoretic verification approach [35]. The witness is confirmed if a violation of the specification is found.

When writing a witness automaton, ULTIMATE AUTOMIZER produces the source-code guards `startline`, `endline`, and `control`, as well as the state-space guard `assumption`. The production of a witness is straightforward, because ULTIMATE AUTOMIZER already computes the necessary information in order to provide human-readable counter-examples.

8. REFERENCES

[1] L. Alawneh and A. Hamou-Lhadj. MTF: A scalable exchange format for traces of high performance computing systems. In *Proc. ICPC*, pages 181–184. IEEE, 2011.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.

[3] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

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

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

[6] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. ICSE*, pages 326–335. IEEE, 2004.

[7] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The BLAST query language for software verification. In *Proc. SAS*, LNCS 3148, pages 2–18. Springer, 2004.

[8] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, pages 25–32. IEEE, 2009.

[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.

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

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

[12] 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.

[13] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. Information reuse for multi-goal reachability analyses. In *Proc. ESOP*, LNCS 7792, pages 472–491. Springer, 2013.

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

[15] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.

[16] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.

[17] D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232. Springer, 2015.

[18] 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.

[19] 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.

[20] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.

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

[22] E. M. Clarke, T. A. Henzinger, and H. Veith. *Handbook of Model Checking*. Springer.

[23] E. M. Clarke, M. Talupur, H. Veith, and D. Wang. SAT-based predicate abstraction for hardware verification. In *Proc. SAT*, LNCS 2919, pages 78–92. Springer, 2003.

[24] E. M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, LNCS 2772, pages 208–224. Springer, 2003.

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

[26] L. Correnson and J. Signoles. Combining analyses for C program verification. In *Proc. FMICS*, LNCS 7437, pages 108–130. Springer, 2012.

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

[28] P. Cousot and R. Cousot. Systematic design of program-analysis frameworks. In *Proc. POPL*, pages 269–282. ACM, 1979.

[29] D. Dams and K. S. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. In *Proc. FMCO*, LNCS 4111, pages 138–160. Springer, 2005.

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

[31] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.

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

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

[34] 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.

[35] 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.

[36] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.

[37] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (OTF). In *Proc. ICCS*, LNCS 3992, pages 526–533. Springer, 2006.

[38] D. Kröning and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *Proc. MEMOCODE*, pages 101–110. IEEE, 2005.

[39] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test-case minimization. In *Proc. ASE*, pages 417–420. ACM, 2007.

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

[41] 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.

[42] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[43] O. Šerý. Enhanced property specification and verification in BLAST. In *Proc. FASE*, LNCS 5503, pages 456–469. Springer, 2009.

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