Proc. TAP 2018, © Springer

Tests from Witnesses

Execution-Based Validation of Verification Results

Dirk Beyer,[1] Matthias Dangl,[1] Thomas Lemberger,[1] and Michael Tautschnig[2]

[1] LMU Munich, Germany [2] Queen Mary University of London, UK

Abstract. The research community made enormous progress in the past years in developing algorithms for verifying software, as shown by international competitions. Unfortunately, the transfer into industrial practice is slow. A reason for this might be that the verification tools do not connect well to the developer work-flow. This paper presents a solution to this problem: We use verification witnesses as interface between verification tools and the testing process that every developer is familiar with. Many modern verification tools report, in case a bug is found, an error path as exchangeable verification witness. Our approach is to synthesize a test from each witness, such that the developer can inspect the verification result using familiar technology, such as debuggers, profilers, and visualization tools. Moreover, this approach identifies the witnesses as an interface between formal verification and testing: Developers can use arbitrary (witness-producing) verification tools, and arbitrary converters from witnesses to tests; we implemented two such converters. We performed a large experimental study to confirm that our proposed solution works well in practice: Out of 18 966 verification results obtained from 21 verifiers, 14 727 results were confirmed by witness-based result validation, and 10 080 of these results were confirmed alone by extracting and executing tests, meaning that the desired specification violation was effectively observed. We thus show that our approach is directly and immediately applicable to verification results produced by software verifiers that adhere to the international standard for verification witnesses.

1 Introduction

Automatic software verification, i.e., using methods from program analysis and model checking to find out whether a program satisfies or violates a given specification, is a successful and mature technology. The efficiency and effectiveness of the available verification tools for C programs is shown in the annual competition on software verification [5]. Despite this success story in research, the state-of-the-art in practice is that not many software projects have such verification tools incorporated into their software-development process. The reason for this gap between availability of technology on the one side and missed opportunities on the other side is perhaps twofold: (a) developers are frustrated by false alarms, i.e., in the past, static analyzers reported too many bugs that

were not observable in a concrete program execution, and thus, developers have lost confidence in bug reports [20]; (b) there is a lack of appropriate interfacing, i.e., it is difficult for developers to leverage advantages of the verification tools because they are difficult to integrate and difficult to learn from [1].

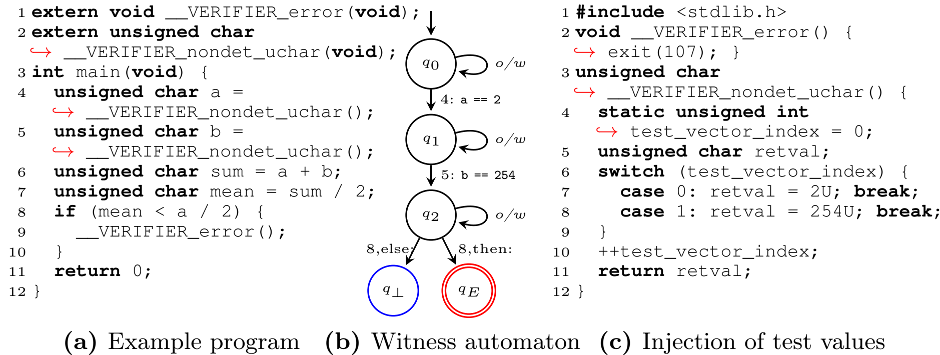
To overcome these two problems, we propose (i) to use verifiers that produce verification witnesses, i.e., abstract descriptions of one or more paths to a specification violation (many such tools are already available [1]), and (ii) to validate whether a real bug has been found by constructing a test from the produced verification witness and observing the execution of that test. This way, issue (a) above is solved because, if the test execution does show and thus confirm the reported specification violation, the verification result can be examined with high confidence and on a concrete, executable example (e.g., with a debugger), and issue (b) is solved because we bridge the gap between the, in most projects, unfamiliar domain of verification and the established domain of testing, which makes it easier to integrate verification into the development process.

Execution-based Validation of Witnesses. Witness validation based on model-checking technology works well [4, 5, 9, 14], but the disadvantage is that due to over-approximation, the validation might be as imprecise as the verification step. A verification witness serves as a (potentially coarse) description of a part of the state space of a program that contains a specification violation, and the witness validators can confirm or reject the error report. We complement the witness-validation technology by direct test execution: A test case (e.g., unit-test code) is built from the violation witness, and this test case provides a precise and transparent way to confirm and examine it.[2] By observing and analyzing an execution that exposes undesirable behavior, developers can convince themselves that the error report is correct, and address the reported bugs without the risk of wasting time on a false alarm. If the execution does not violate the specification, the witness might have represented a false alarm and the developer can assign a lower priority to that report.

Witnesses as Communication Interface. One barrier for the adoption of verification technology is that developers have to spend considerable time on understanding a verification tool and on becoming familiar with it. Thus, we have to avoid the "lock-in" effect: people might not want to decide for one particular tool if they have to invest time again when they wish to change the decision later. If the developer constructs the integration on top of the exchangeable verification witnesses, i.e., using the witnesses as interface to the verification tools, the verification tool is exchangeable without any change to the testing process.[3]

Tests from Witnesses. In order to flexibly bridge the gap from witness to test, we provide two independently developed implementations of tools that take as input a program and a violation witness, and synthesize a test that is compilable

[1] `https://sv-comp.sosy-lab.org/2017/systems.php`
[2] It has been shown that model checkers can be effective in constructing useful tests [12].
[3] At least 21 verifiers are available that produce witnesses in the exchangeable format (cf. Table 1, which lists the verifiers that we use in our experiments).



(a) Example program **(b)** Witness automaton **(c)** Injection of test values

Fig. 1: An incorrect example C program (a), the corresponding violation witness produced by the verifier (b), and a code fragment used to inject the extracted test values for compilation (c)

and executable. This approach provides the following three features: (1) the result of a verification tool can be validated by compiling and executing the corresponding test—if the test violates the specification, the verification tool reported a correct alarm and the result can be handled appropriately; (2) the synthesized unit tests can be stored and maintained together with the other unit tests, but can also be re-constructed at any time on demand; (3) independently from the verification tool that produced the witness, the full repertoire for inspecting a failing program—such as debuggers, profilers, and visualization tools—can be used by the developer to understand the bug that the test represents.

Experimental Study. To evaluate our proposal, we performed experiments on thousands of witnesses. We took many C programs from the largest public repository of verification tasks and many witness-producing verification tools, and collected 13 200 witnesses of specification violations. We obtained another 5 766 refined witnesses using witness refinement, a procedure introduced in the original work on verification witnesses [9]. This technique is supposed to refine witnesses to be more concrete, so we should be able to generate better test cases from them. In conjunction with the two existing validators, CPAchecker and Ultimate Automizer, our method significantly increases the confirmation rate: out of the total of 18 966 witnesses, we were able to extract test cases for 10 080 of them, meaning that we successfully created and executed the tests, and the specification violation was observed. Using the new approach, we increased the confirmed results from 12 821 to 14 727 in total.

Example. In the following, we illustrate the complete process from running a verification task using a verifier through synthesizing the test code from the violation witness to compiling the program and executing it.

Fig. 1a shows a program that attempts to calculate the mean of two integer numbers, a computation that is often required in binary-search algorithms.

[4] The example also works for larger data types, but for ease of presentation, we aim to keep the range of values small, so that all calculations can be followed by hand.

In lines 4 and 5, two variables `a` and `b` of type `unsigned char` [4] are initialized nondeterministically, for example from user input. The subsequent lines are supposed to calculate the mean of the two variables, by first computing their sum in line 6 and then dividing it by `2` in line 7. If the mean of `a` and `b` has been calculated correctly, it must not be less than half of either of the two values. This condition is asserted in lines 8 to 10. We can check whether the condition is satisfied by specifying that the function `__VERIFIER_error()` must not be reachable, and then running a verifier on this verification task. The verifier should detect and report that the assertion will be violated if the sum of `a` and `b` exceeds the range of the data type `unsigned char`, causing an overflow. Fig. 1b shows a violation-witness automaton [9] that represents a counterexample to the specification. The automaton specifies that if we assume that `a` is assigned the value `2` in line 4 and `b` is assigned the value `254` in line 5, control will flow to the `then`-branch in line 8, causing a violation of the specification. To independently validate this witness, we can then extract the input values for `a` and `b`, and use them to provide an implementation of the input function `__VERIFIER_nondet_uchar()` and the `__VERIFIER_error()` function as depicted in Fig. 1c. After compiling Fig. 1a and 1c into an executable and running it, we can confirm that these input values trigger the call to `__VERIFIER_error()` by checking its return code. We can even use a debugger such as GDB to step through the compiled program and observe the faulty behavior directly. The debugger will show that the sum of `a` and `b`, respectively `2` and `254`, computed in line 6 wraps around to `0`. Therefore, the mean is incorrectly calculated as `0` in line 7. The condition in line 8 then evaluates to `1`, because `0` is smaller than `1`.

It must be noted that the witness depicted in Fig. 1b is very precise: it provides a concrete counterexample with explicit values for `a` and `b`. But in general, a violation witness may simply describe a part of the state space that contains a specification violation, i.e., an abstract counterexample. Suppose a verifier is only able to provide a witness that specifies that if `a + b` is greater than `255` in line 6, the specification will be violated. By using witness refinement [9], we can obtain from this abstract witness a concrete witness like Fig. 1b.

Contributions. Our approach features the following advantages:

- Verification tools sometimes produce false alarms, which can lead to severe waste of investigation time. We synthesize tests from verification witnesses, and consequently trust only verification results confirmed by test execution.
- There are several witness-based validators available, but our execution-based validation of the error path can be more precise and more efficient, compared to the previously available validators.
- Avoidance of technology lock-in: A developer's work flow does not depend on a particular choice of verification tool, because the developer's infrastructure hooks in at the witness. The developer may elect to use a different verifier, or even use multiple verifiers simultaneously—at no additional cost.
- Compared to working with witnesses, developers are more familiar with tests, and more supporting tools—such as profilers, memory analyzers, and visualization tools—are available to analyze the tests that correspond to the witnesses.

– The newly generated tests can complement the existing test suite, and the tests as well as the witnesses can be stored and maintained as first-class objects in the software life cycle.

Related Work. Our approach is based on a number of existing ideas, which we outline in the following.

Verification Witnesses. We build our contributions on top of existing work on violation witnesses [9], which we will describe in more detail in the background section. The problem that verification results are not treated well enough by the developers of verification tools is known and there are also other works that address the same problem, for example, the work on execution reports [18].

Test-Case Generation. The idea to generate test cases from verification counterexamples is more than ten years old [6, 48], has since been used to create debuggable executables [39, 42], and was extended and combined to various successful automatic test-case generation approaches [25, 27, 36, 46]. We complement existing techniques in the following ways: Our technique works on the flexible exchange format for violation witnesses. In case such a witness constitutes only an abstract counterexample, we can use witness refinement to efficiently obtain a concrete one [9]. Such a mechanism is not available for existing test-case generation tools.

Execution. Other approaches [16, 22, 35] focus on creating tests from concrete and tool-specific counterexamples. In contrast, our approach does not require full counterexamples, but works on more flexible, possibly abstract, violation witnesses.

Debugging and Visualization. Besides executing a test, it is important to understand the cause of the error path, and there are tools and methods to debug and visualize program paths [3, 7, 28].

2 Background

A verification witness is an exchangeable object that stores valuable information about the verification process and the verification result. The key is that the format is open and exchangeable, and that many verification tools support it.

Witness Construction. It has been commonly established practice for verifiers to provide a counterexample to witness a specification violation, in particular since counterexamples were used to refine abstract models [21]. The problem was that these counterexamples were more or less 'dumps' of paths through the state space, sometimes not human-readable, sometimes not machine-readable. Recent efforts of the software-verification community established a common exchange format for verification results as verification witnesses [9]. In this format, a so-called violation-witness automaton (as seen in Fig. 1b) describes a state space that contains the specification violation. This state space does not necessarily have to represent just a single error path, but may contain multiple error paths and even paths without a specification violation. As an example for the use of verification witnesses, the International Competition on Software Verification (SV-COMP) applies this format and counts a report of a found bug only if a

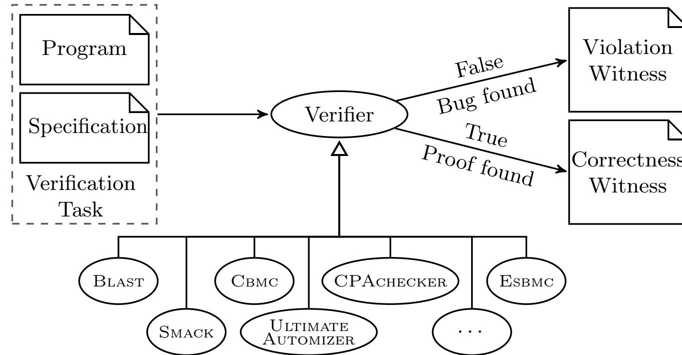
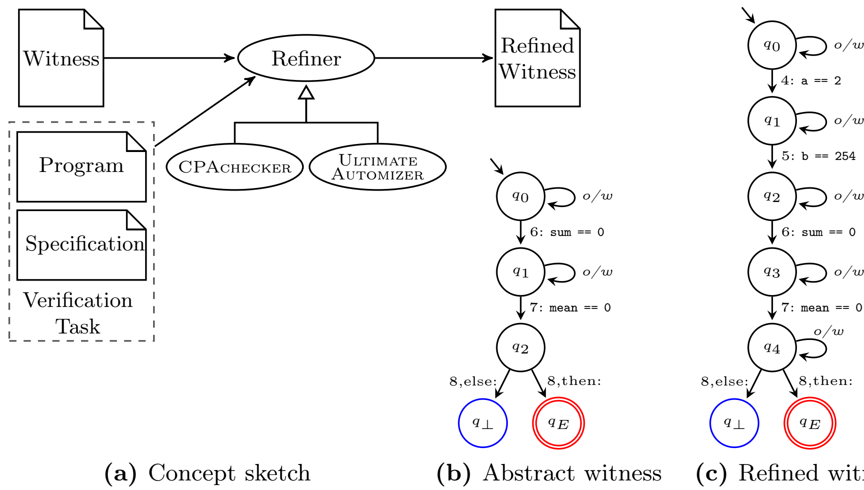


Fig. 2: Software verifiers produce witnesses



(a) Concept sketch **(b)** Abstract witness **(c)** Refined witness

Fig. 3: Concept of witness refinement with example abstract and refined witnesses for the example program depicted in Figure 1a from the introduction

corresponding violation witness is reported and confirmed [4]. Figure 2 illustrates the process: the verifiers can be exchanged according to the needs of the user, there is no risk of technology lock-in. Figure 2 also shows that the exchange format for witnesses has recently been extended to correctness witnesses [8]. In the remainder of this paper, however, we will only consider violation witnesses.

Witness Refinement. The original work on verification witnesses [9] contains the proposal to consider refinement of witnesses. The idea is to take a violation witness as input, replay it with a validating verifier, and produce a new witness that is more detailed. A more detailed violation witness is closer to a concrete program path and makes the validation process faster. We will later in this paper use an instance of a witness refiner to improve witnesses from other verification

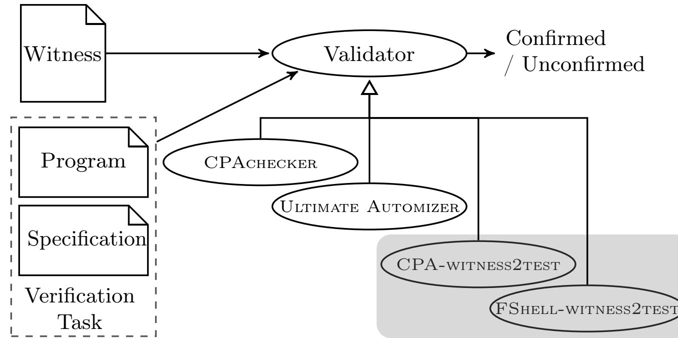


Fig. 4: Violation-witness validation

tools towards being able to successfully derive tests from witnesses. Figure 3a illustrates the optional step of using witness-refining validators to strengthen a witness. Figure 3b shows another, valid violation witness for the previously considered program from Figure 1a. In contrast to the witness in Fig. 1b, this witness does not specify any concrete values for the two nondeterministic values of variables `a` and `b`, but specifies that a property violation occurs if the intermediate variables `sum` and `mean` are both equal to `0`. This witness automaton represents a set of 256 different counterexamples: every counterexample with values for `a` and `b`, so that `a + b == 0` during execution. Figure 3c shows a violation witness that is a refinement of the more abstract witness in Fig. 3b that additionally specifies concrete values for the two variables `a` and `b` and thus restricts the search space in witness validation early on.

Witness Validation. Violation witnesses can be used to independently re-establish the verification result by using a witness-based result validator that takes the information from the witness to find a path through the state space of the program to a specification violation. Thus, a successful validation increases trust in the verification result, and developers no longer need to rely on the verifiers alone. Instead, they can focus their attention on the validated results and assign a lower priority to unconfirmed alarms. The existing witness-based result validators employ potentially-expensive model-checking techniques to replay error paths that are represented in the witness. While this is a powerful technique (it can reconstruct error paths even for abstract witnesses), the technique still has the limitations of common program-analysis and model-checking techniques, namely that the technique may over-approximate the semantics of the programming language, thus potentially confirming false alarms or rejecting valid violation witnesses. As a solution to this, we propose an execution-based approach to witness-based result validation. Figure 4 shows the two existing validators CPAchecker and Ultimate Automizer together with the two new, execution-based validators that we introduce in this paper: CPA-witness2test and FShell-witness2test.

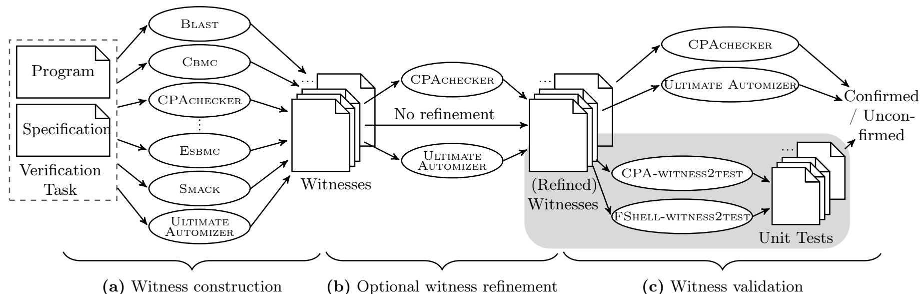


Fig. 5: Software verification with witnesses: construction, (optional) refinement, and validation work flow

3 Tests from Witnesses

This section introduces a new, yet unexplored, application of witnesses that can easily be integrated into established processes for verification-result validation, as summarized by Fig. 5. The highlighted area in Fig. 5 outlines the goal: for a given violation witness, we want to construct a test that can be compiled and executed to check that the bug is realizable. In particular, driven by our desire to keep the work-flow independent from special verifiers, we want to have two independently developed implementations of such witness-to-test tools.

Our new, execution-based witness validator does not require the aid of model-checking techniques for validating verification results: we generate a test harness (test code for the program), which can be compiled and linked together with the original subject program and executed. If the execution does not trigger the described bug, the witness is deemed spurious, i.e., not realizable.

Adding this new tool to the pool of available witness-based result validators not only increases the diversity of validation techniques and its potential for establishing trust in verification results, but also adds novel features to the validation process: As a valuable by-product of a successful validation, the developers are able to obtain executable test code that is guaranteed to reproduce the bug in their system, and they can use all of the infrastructure for inspecting and debugging that they are trained and experienced in and that is already in place in their development environment. For example, a C developer might simply run GDB to step through the executable error path.

Figure 6 shows the complete picture of execution-based witness validation. The verification task (a given program with a given specification) is verified by a chosen verifier. If the verifier reports a specification violation (False, bug found) it also produces a violation witness. (Our work does not consider the outcome True, for which the development of practical support, such as correctness witnesses [8] and compact proof witnesses [32], is also a subject of ongoing research.) The witness in GraphML format [15] is then given to witness2test, which synthesizes a test harness that drives the program to the specification violation. In order to support our claim of independence from any particular tool implementation, we implement two completely different instances

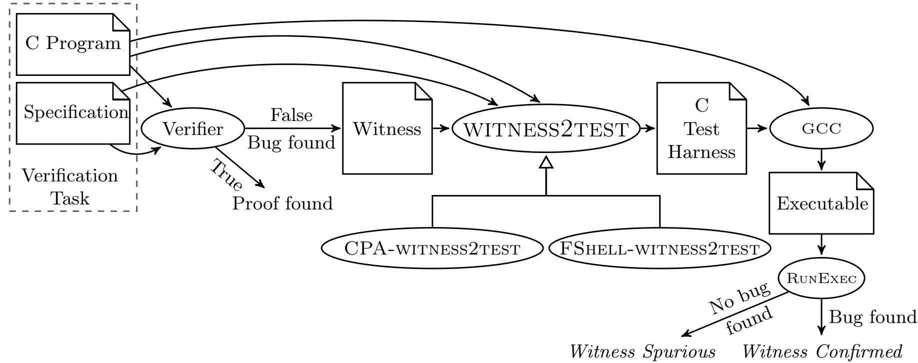


Fig. 6: Flow of execution-based result validation

of witness2test, namely CPA-witness2test (based on open-source components from CPAchecker) and FShell-witness2test (based on ideas from FShell). The test-harness and the original (unchanged) program are then compiled and linked to obtain an executable program. The executable program is then executed in a safe execution container.[5] If the reported specification violation is observed during this execution, the witness is confirmed. Otherwise the witness is not confirmed, most likely because the witness is not precise enough or even spurious.

3.1 CPA-witness2test

One of our implementations for the witness2test component of the architecture outlined in Fig. 6 is CPA-witness2test, which is based on the CPAchecker framework [11]. For our purpose of matching an input witness to the program source code of a verification task and generating a *test harness*, we configure CPAchecker to use the witness automaton as a *protocol automaton* [9] to guide and restrict the state-space exploration to the program paths that the witness represents. Unlike observer automata [44], which we use to represent the specification and which can only monitor the state-space exploration of an analysis, *protocol automata* may also restrict the state-space exploration, for example to a specific program path, thereby guiding the analysis along that path. In our case, this path is the error path represented by the protocol automaton. We configure the analysis to only consider the (syntactical) branching information of the *protocol automaton* and to not semantically analyze the path. During this *protocol analysis*, we observe which input-value assumptions from the witness correspond to which input function or variable of the program. By collecting this information, we are able to construct a *test vector* for the program. The *test vector* maps an input value to each input variable and a list of input values to each external function. We synthesize a *test harness* from a test vector by providing initializations for input variables and definitions for external functions. An external function with a list $(v_0, \ldots, v_{n-1})$ of $n \in \mathbb{N}$ input values is defined by using a `switch` statement with n cases over a static counter variable $0 \leq i < n$ that is initialized to 0 and incremented after each call to the function. Each case of the

[5] We choose BenchExec [13] as container solution, because it is also used by SV-COMP.

`switch` statement corresponds to an input value, such that `case` i selects v_i. We also inject a call to the `exit` function so that when we later execute the program, we can detect that the intended violation of the specification was triggered, i.e., the program crashed precisely due to the bug described by the witness, by checking for a specific execution return value. Fig. 1c shows the `exit(107)`-call in line 2 and a definition of an input function `__VERIFIER_nondet_uchar()` in lines 3 to 12 as generated by CPA-witness2test, where the counter variable `test_vector_index` represents i. The `switch` statement in this function definition provides sequential access to the two input values $(2, 254)$ that CPA-witness2test extracted from the witness of Fig. 1b for the program shown in Fig. 1a.

3.2 FShell-witness2test

The key design principle of FShell-witness2test is independence from existing verification infrastructure: FShell-witness2test's results shall—by design—be unbiased towards any existing software-analysis framework. While this does imply limitations on the class of witnesses that can be processed as discussed below, it does yield further advantages: FShell-witness2test is easy to extend for prototyping, and does not require any background in software verification.

FShell-witness2test comprises two major parts: (1) A Python-based processor of the witness and the input program, using `pycparser`[6] to generate test vectors in a format compatible with FShell [31]. (2) A Perl script that translates such test vectors into a test harness.

For a given verification task and witness, FShell-witness2test first parses the specification to restrict itself to reachability properties (call to error function should not be reachable). The witness and the C program are then handed to the Python-based processor. The specification defines the entry function to be used by the generated test harness.

As `pycparser` cannot handle various GCC extensions, input programs are preprocessed and sanitized by performing text replacement and removal. We then obtain the abstract syntax tree and iterate over its nodes to gather data types and source locations of (1) all procedure-local uninitialized variables, (2) all functions with prefix `__VERIFIER_nondet`, and (3) all uses of such functions. We refer to the locations of uninitialized variables and nondeterministic-input function uses as *watch points*.

Finally we build a linear sequence of nodes from the GraphML encoding of the witness. Traversing this sequence, any match of line numbers against the watch points triggers an attempt to extract values from assumptions in the witness. If parsing the C code that is contained in the assumption succeeds, then an input value is recorded.

The test vector is compatible with the output of FShell; the program of Fig. 1 yields the following test vector:

```
IN:
  ENTRY main()@[file mean.c line 1]
  unsigned char __VERIFIER_nondet_uchar()@[file mean.c line 4]=2
  unsigned char __VERIFIER_nondet_uchar()@[file mean.c line 5]=254
```

[6] `https://github.com/eliben/pycparser`

Such a test vector is translated to a Makefile that generates an actual test harness, which consists of invocation code and the implementation of various nondeterministic-input functions that are present in the program. FShell-witness2test reports False (confirming the violation) if, and only if, the property violation is detected in the output of the test execution.

4 Evaluation

We perform a large experimental study to demonstrate the general applicability and the advantages of our approach.

4.1 Evaluation Goals

The goal of our experimental evaluation is to collect experience with our new kind of result validation and to support the following claims with data for a large set of witnesses:

Claim 1: Execution-based validators can confirm violation witnesses that the existing validators (which are based on model-checking technology) can not validate. Thus, execution-based validation increases the overall effectiveness.

Claim 2: Result validation based on executable tests can be faster than result validation based on model-checking technology.

Claim 3: Violation witnesses in the common exchange format for verification results (cf. Sect. 2) are a valuable source to synthesize test code for specification violations to complement existing test suites.

4.2 Experiment Setup

We used the benchmarking framework BenchExec (revision `fb32a3e7`) to conduct our experiments. In order to experimentally evaluate our approach, we first construct a large set of witnesses that is diverse in terms of (a) subject programs and (b) verification tools that create witnesses.

Subject Programs. For (a), we consider the largest available set of verification tasks [7] from the community of automatic software verification and select all 5 692 verification tasks with a reachability property [8].

Verifiers. For (b), we use all verification tools that participated in SV-COMP 2017 for property *ReachSafety* and whose license allows us to use it [9]. Table 1 lists all verifiers that we executed to produce violation witnesses. The table lists in the first column the verifier name with a link to the project web site for more information, and a reference to the paper describing the corresponding verifier. For the experiments, we took the archives from the competition web site. [10]

[7] `https://github.com/sosy-lab/sv-benchmarks/tree/423cf8c`

[8] We have to restrict the experiments to property *ReachSafety* because there were no witness validators available for the other properties.

[9] There are also two commercial verifiers that produce witnesses, but we cannot use them due to their proprietary license.

[10] `https://sv-comp.sosy-lab.org/2017/systems.php`

Table 1: Violation witnesses produced by verifiers and resulting tests

Verifier		Produced witnesses			Produced tests			
		Unref.	Ref.	Total	Count	kLOC	kB	# Inputs (Avg.)
2LS	[45]	992	384	1 376	1 208	89.9	3 999	7.57
Blast	[47]	778	202	980	327	29.0	938	0.271
Cbmc	[34]	831	467	1 298	1 249	67.7	2 991	6.33
Ceagle		619	426	1 045	540	92.2	262	5.39
CPA-BAM-BnB	[2]	851	175	1 026	158	42.9	1 114	0
CPA-kInd	[10]	263	193	456	656	56.2	2 967	14.9
CPA-Seq	[23]	883	767	1 650	838	95.5	3 895	1.79
DepthK	[43]	1 159	305	1 464	1 302	65.4	3 170	2.96
Esbmc	[37]	653	148	801	478	21.0	1 983	2.53
Esbmc-falsi	[37]	981	395	1 376	1 133	53.7	1 906	1.81
Esbmc-incr	[37]	970	392	1 362	1 126	53.5	1 896	1.82
Esbmc-kInd	[24]	847	352	1 199	1 028	48.9	1 774	1.69
Forester	[30]	51	0	51	0	0	0	-
PredatorHP	[33]	86	61	147	80	17.2	434	0
Skink	[17]	30	25	55	44	0.290	8	0
Smack	[41]	871	632	1 503	1 576	128	5 654	6.09
Symbiotic	[19]	927	411	1 338	589	38.1	1 375	0
SymDIVINE	[38]	247	224	471	405	13.4	580	0
UAutomizer	[29]	514	70	584	121	2.24	59	0
UKojak	[40]	309	67	376	116	2.15	55	0
UTaipan	[26]	338	70	408	121	2.23	59	0
Total		13 200	5 766	18 966	13 095	920	35 119	5.60

Collection of Witnesses. From the given verification tasks and verifiers, we started verification runs and collected the obtained violation witnesses. For this replication of the SV-COMP experiments we followed thoroughly the description on the competition web site [10] and in the report [4]. In particular, we started each verifier only on those verification tasks and with those parameters that were declared by the development teams of the verifiers[11]. The number of witnesses that we obtained with this process is reported in Table 1 (col. 'Unref.'). Because we use all available verifiers (not only those that performed well in the competition), the set of witnesses contains also bad witnesses (e.g., that are syntactically incorrect). We did not want to exclude them for external validity.

To further increase the external validity of our evaluation, we additionally produced witnesses by applying a witness-refinement technique (cf. Sect. 2) to 13 200 witnesses above. We used the witness-refiner from the CPAchecker framework for this step. This refinement is often able to improve imprecise witnesses by adding concrete input values, and yields another 5 766 witnesses (col. 'Ref.') to a total of 18 966 witnesses (col. 'Total') that we will run our experiments on.

In order to highlight the differences between model-checking-based validation approaches and execution-based validation approaches, we manually crafted some verification tasks and corresponding witnesses. These witnesses

[11] `https://github.com/sosy-lab/sv-comp/tree/svcomp17/benchmark-defs`

Table 2: Confirmed witnesses and verification results

	Static validators			Dynamic validators			Union
	CPAchecker	Automizer	Union	CPA-w2t	FShell-w2t	Union	
Confirmed witnesses	11 225	7 595	12 821	7 151	7 545	10 080	14 727
Unref. witnesses	5 750	3 450	7 214	3 506	3 459	5 082	9 056
Ref. witnesses	5 475	4 145	5 607	3 645	4 086	4 998	5 671
Incorrectly confirmed	18	7	25	6	0	6	31
Confirmed verif. results	5 751	5 643	7 215	5 377	5 755	7 292	9 057
Incorrectly confirmed	15	7	22	6	0	6	22

allow us a more detailed discussion of some effects, but were not added to our set of automatically generated witnesses.

Computing Resources. Our experiments were conducted on machines with an Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 16.04 with Linux kernel 4.4). We limited the verification runs to four processing units (i.e., two physical cores), 7 GB of memory, and 15 min of CPU time, and the witness-refinement and validation runs to two processing units (i.e., one physical core), 4 GB of memory, and 1.5 min of CPU time. All CPU times are reported with two significant digits. The limits are inspired by SV-COMP.

Validators. We used CPA-witness2test in revision 24 475 from CPAchecker and FShell-witness2test in revision `2a76669f` from the `test-gen` branch. We used the model-checking based witness validators CPAchecker, revision 24 475, and Ultimate Automizer 0.1.8.

4.3 Availability of Data and Tools

All tools and all data obtained in our experiments are available via our supplementary web page.[12] The verification tasks are also publicly available.[7]

4.4 Results

Claim 1: Effectiveness. Table 2 reports the number of witnesses that the individual validators were able to confirm. In the columns, it shows: the results of the static validators CPAchecker and Ultimate Automizer, as well as the union of these two; the results of the dynamic validators CPA-w2t and FShell-w2t, as well as the union of these two; and the results of the union of all four validators. The union is the number of witnesses that at least one of the considered validators was able to confirm, i.e., one of CPAchecker and Ultimate Automizer (col. 4), or one of CPA-w2t and FShell-w2t (col. 7), or any of the four (col. 8). In the rows, Table 2 is divided into confirmed witnesses (unrefined and refined witnesses, as well as incorrectly confirmed witnesses) and confirmed verification results. A witness is incorrectly confirmed if the verification result reported by a verifier is wrong and the validator reached the same, wrong conclusion using the verification-result witness that was provided by the verifier. Since for each unrefined witness from a verifier, a refined counterpart may exist, the number of confirmed witnesses is potentially double the number of verification

[12] `https://www.sosy-lab.org/research/executionbasedwitnessvalidation/`

Table 3: Performance comparison for witnesses that all validators confirmed (CPU time for 2 685 witnesses)

	CPAchecker	Automizer	CPA-w2t	FShell-w2t
Total time (s)	20 000	45 000	30 000	1 900
Average time (s)	7.4	17	11	0.72
Median time (s)	6.2	11	5.9	0.71

results that were confirmed using these witnesses. Because of this, Table 2 also reports the number of confirmed verification results. We considered a verification result as confirmed if at least one of its witnesses is confirmed by the used validators. This can be the unrefined witness, or, if it exists, the refined one. The results of Table 2 show that the static validators together confirmed a total of 12 821 verification results, while the dynamic validators together confirmed a total of 10 080 results. Also, the two different validation techniques confirm different results: a union of 14 727 results were confirmed by both validation techniques together. Of the verification results that neither of the static validators was able to confirm, CPA-w2t was able to confirm 735 and FShell-w2t was able to confirm 1 488, meaning that the techniques complement each other well. Together, they were able to confirm 1 842 results that no static validator was able to confirm. This shows that the independently developed dynamic techniques complement each other because they are based on completely different technology. It is also interesting to consider wrong witnesses, i.e., violation witnesses that constitute false alarms. In our experiments, the verifiers produced 679 false alarms. Of these, the static approaches incorrectly confirmed 22 wrong witnesses (of different programs), while FShell-w2t did not wrongly confirm any false alarms. CPA-w2t confirmed 6 wrong witnesses incorrectly, all based on programs that contain floating-point arithmetic. For these, CPA-w2t has only limited support. Despite that, this highlights a high precision of our execution-based approach. In sum, using dynamic validators in addition to static validators can significantly increase the number of successfully validated verification results.

Claim 2: Efficiency. Table 3 considers only results that were confirmed by all validators, to compare the execution performance. For the dynamic validators, the reported run time contains all three steps: generating the test from the witness, compiling and linking, and executing the test. The results show that the static approaches are slow (CPAchecker and Ultimate Automizer), that the approach that assembled a static analysis for test generation from CPAchecker components is also slow (CPA-w2t), and that the light-weight implementation that is specifically tailored to generating tests from witnesses is extremely fast (FShell-w2t). Figure 7 displays quantile functions that show for each validator the necessary maximal CPU time (y-axis) for confirming a certain quantile of results (x-axis). We observe that FShell-w2t significantly outperforms all other validators.

Interestingly, in our validation we observed that the witnesses that require the most time to validate are witnesses that are large in size and that describe a long, detailed error path. Most of these are produced by verifiers that use bounded model checking, e.g., Cbmc and CPA-kInd, or by our refinement step.

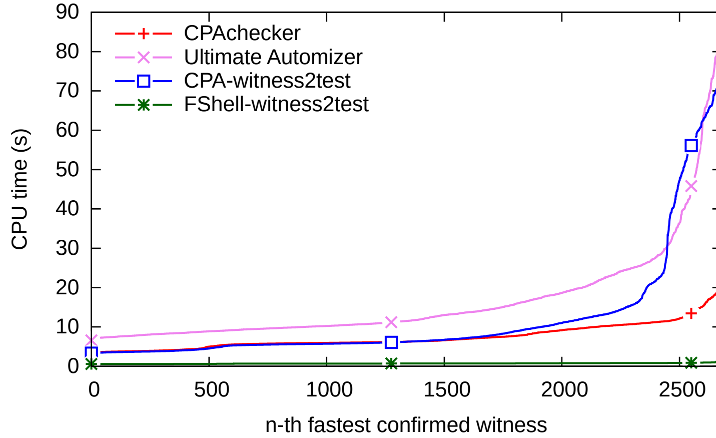
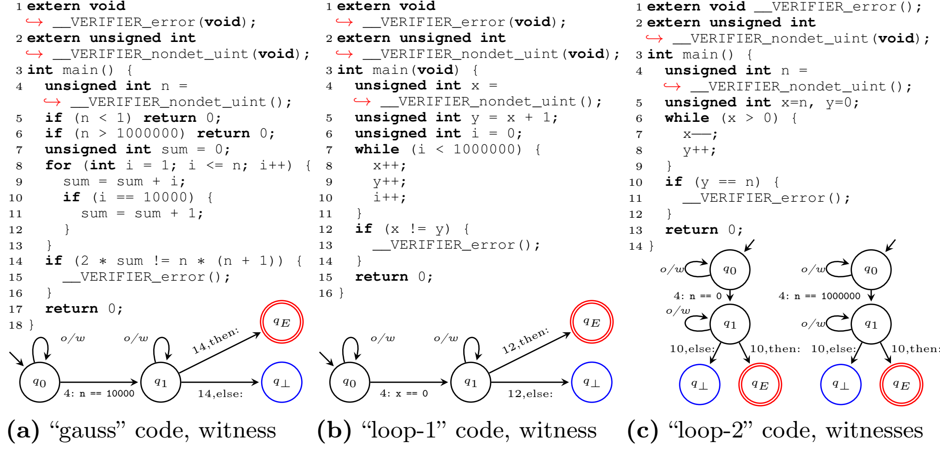


Fig. 7: Quantile plot for CPU time consumed for validating witnesses accepted by all validators

Claim 3: Test Generation. The last four columns of Table 1 relate the number of witnesses that we processed to the number of produced tests for which failing executions are realizable. With 'produced tests' we refer to the tests that were produced by any of the dynamic validators and for which the test execution lead to an observed specification violation. Note that because we collect tests from both dynamic validators, the numbers of produced tests exceed the number of witnesses in some rows. Since the tests are available in source code, and could be maintained and re-used by developers in practical application scenarios, we also report the size of these unit tests in lines of code, file size, and the average number of input values per generated unit test. The table shows that the number of unit tests and the accompanying size of test code that the approach can produce are significant. The results confirm that we are able to provide an interface to verification tools via witnesses and tests that avoids technology lock-in and which enables developers to explore the verification results using tools and techniques they are familiar with. The combination of software verification and execution-based result validation may also be used to automatically extend the existing test suites of a project.

4.5 Detailed Discussion of Synthetic Examples

Now we discuss a few effects in more detail on hand-crafted example witnesses. Bugs that occur after only few loop iterations are also known as *shallow* bugs, as opposed to *deep* bugs that occur after many loop iterations. One of the strengths of dynamic validation approaches is that long loops can simply be executed, while model checkers usually need to perform expensive symbolic unrolling to reveal deep bugs, which is therefore a more difficult task for them than discovering shallow bugs. Thus, we expect the set of witnesses obtained from model checkers to consist mostly of shallow bugs, while at the same time we must expect that the advantages of test-based validation become most apparent for witnesses for



(a) "gauss" code, witness **(b)** "loop-1" code, witness **(c)** "loop-2" code, witnesses

Fig. 8: Hand-crafted tasks and witnesses

deeper bugs, which necessitate many unrollings. Therefore, we hand-crafted a small set of verification tasks and witnesses, including the example for computing the mean from Fig. 1a in the introduction, to exemplify the differences between the test-based approaches and those based on model checking.

Fig. 8a shows an example program intended to compare the iterative sum of ascending values with the result of the Gauss sum formula, and a witness for a bug in the program. The bug is located in lines 10 to 12 and causes an error for inputs larger than or equal to 10 000. The depicted witness for this bug assigns an input value of 10 000. Fig. 8b shows an example program that increments two variables `x` and `y` 1 000 000 times and then asserts their equality in line 12, and a witness for a violation of this assertion. Since `y` is initialized to `x + 1` in line 5, the assertion will fail for any value of `x`. The depicted witness for this bug assigns an input value of 0. Fig. 8c shows an example program with a variable `n` initialized with an input function in line 4 and copies its value to a variable `x` in line 5. In the same line, a variable `y` is initialized to 0. Then, in lines 6 to 9, `x` is decremented and simultaneously `y` is incremented, until `x` is 0, so essentially, `y` counts the loop iterations, and $n - x = y$ is a loop invariant. Consequently, `y` must be equal to `n` at the end of the loop, and therefore the call to the error function in line 11 is called for any input value, so that both witnesses in Fig. 8c are valid counterexamples. The first of these witnesses, however, describes a violation that skips the loop entirely with an input value of 0, while the second one, due to assigning an input value of 1 000 000, reaches the violation in line 11 only after 1 000 000 loop iterations. We expect all validators to quickly validate the witnesses for shallow bugs, i.e., the one depicted in Fig. 1a and the first witness in Fig. 8c, but we expect test-based validators to perform significantly better on the witnesses for deep bugs, i.e., those depicted in Figs. 8a and 8b, and the second witness in Fig. 8c. Table 4 reports the results for validating these tasks and largely confirms our expectations. While CPAchecker exceeds its resource

Table 4: Validation of hand-crafted witnesses

Witness	CPAchecker		Automizer		CPA-w2t		FShell-w2t	
	Result	Time (s)	Result	Time (s)	Result	Time (s)	Result	Time (s)
gauss	M	-	✗	11	✓	3.4	✓	0.60
loop-1	T	-	✗	9.6	✓	3.4	✓	0.60
loop-2/wit-1	✓	3.8	✓	8.0	✓	3.4	✓	0.58
loop-2/wit-2	T	-	✓	7.5	✓	3.2	✓	0.58
mean	✓	3.5	✓	7.1	✓	3.6	✓	0.58

limitations ("M" for exceeding the memory limit, "T" for exceeding the CPU time limit) for all witnesses except for the two that represent shallow bugs, CPA-w2t and FShell-w2t quickly confirm all witnesses (✓). It is somewhat surprising to see that Ultimate Automizer is able to confirm the `loop-2/wit-2` of Fig. 8c. Checking the tool output, however, reveals that Ultimate Automizer ignored the input value of `n` specified by the witness and used 0 instead of 1 000 000. We were also surprised that the witnesses in the first two rows were rejected by Ultimate Automizer (✗), but since the confirmations of the execution-based validators along with their trustworthy executable tests give us confidence that the witnesses are correct, we assume that the rejections are either caused by the complexity of validating the witnesses or by an approximating behavior of Ultimate Automizer similar to the one leading to the rejection of `loop-2/wit-2`. Overall, we confirm that for this class of witnesses, dynamic approaches are more efficient and more effective than static approaches.

5 Conclusion

Developers are familiar with testing, and there are many tools available for bug analysis that are based on execution, such as debuggers. We try to close the gap between available verification tools and the desire for more precise bug finding by leveraging verification witnesses in an exchangeable standard format. We synthesize tests (test code) from verification results (witnesses) and check the tests for realizability by compiling them, linking them together with the original program, and executing the result in an isolating container. Prior to our work, developers would execute a verification tool and obtain the verification results, which include a violation witness in case a bug is found. Now, we can use the violation witness to obtain a test that drives the program to the specification violation (i.e., into the crash that the developer wants to investigate), while at the same time, we avoid verification-tool lock-in due to the exchangeable standard format. The approach reports only those tests to the developer that really expose the bug; any false alarms are suppressed. The results of our thorough experimental study are encouraging: We verified thousands of programs from the largest publicly-available collection of C verification tasks, consisting of 73 million lines of source code (2.3 GB), and synthesized tests that confirmed 7 286 verification results exposing known bugs in 974 different verification tasks.

References

1. J. Alglave, A. F. Donaldson, D. Kroening, and M. Tautschnig. Making software verification tools really work. In *Proc. ATVA*, LNCS 6996, pages 28–42. Springer, 2011.
2. P. Andrianov, K. Friedberger, M. U. Mandrykin, V. S. Mutilin, and A. Volkov. CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 355–359. Springer, 2017.
3. C. Artho, K. Havelund, and S. Honiden. Visualization of concurrent program executions. In *Proc. COMPSAC*, pages 541–546. IEEE, 2007.
4. D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In *Proc. TACAS*, LNCS 9636, pages 887–904. Springer, 2016.
5. D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017.
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 and M. Dangl. Verification-aided debugging: An interactive web-service for exploring error witnesses. In *Proc. CAV*, LNCS 9780, pages 502–509. Springer, 2016.
8. D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: Exchanging verification results between verifiers. In *Proc. FSE*, pages 326–337. ACM, 2016.
9. 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.
10. D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. CAV*, LNCS 9206, pages 622–640. Springer, 2015.
11. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
12. D. Beyer and T. Lemberger. Software verification: Testing vs. model checking. In *Proc. HVC*, LNCS 10629, pages 99–114. Springer, 2017.
13. D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer*, 2017.
14. 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.
15. 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.
16. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proc. CCS*, pages 322–335. ACM, 2006.
17. F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. G. de Aledo Marugán. Skink: Static analysis of programs in LLVM intermediate representation (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 380–384. Springer, 2017.
18. R. Castaño, V. A. Braberman, D. Garbervetsky, and S. Uchitel. Model checker execution reports. In *Proc. ASE*, pages 200–205. IEEE, 2017.

19. M. Chalupa, M. Vitovská, M. Jonás, J. Slaby, and J. Strejcek. Symbiotic 4: Beyond reachability (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 385–389. Springer, 2017.
20. M. Christakis and C. Bird. What developers want and need from program analysis: an empirical study. In *Proc. ASE*, pages 332–343. ACM, 2016.
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. C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. ICSE*, pages 422–431. ACM, 2005.
23. M. Dangl, S. Löwe, and P. Wendler. CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 423–425. Springer, 2015.
24. M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro. Handling loops in bounded model checking of C programs via k-induction. *STTT*, 19(1):97–114, 2017.
25. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.
26. M. Greitschus, D. Dietsch, M. Heizmann, A. Nutz, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. Ultimate Taipan: Trace abstraction and abstract interpretation (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 399–403. Springer, 2017.
27. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proc. FSE*, pages 117–127. ACM, 2006.
28. E. L. Gunter and D. A. Peled. Path exploration tool. In *Proc. TACAS*, LNCS 1579, pages 405–419. Springer, 1999.
29. M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, A. Nutz, B. Musa, C. Schätzle, C. Schilling, F. Schüssele, and A. Podelski. Ultimate Automizer with an on-demand construction of floyd-hoare automata (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 394–398. Springer, 2017.
30. L. Holík, M. Hruska, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. Forester: From heap shapes to automata predicates (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 365–369. Springer, 2017.
31. A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. How did you specify your test suite. In *Proc. ASE*, pages 407–416. ACM, 2010.
32. M.-C. Jakobs and H. Wehrheim. Compact proof witnesses. In *Proc. NFM*, LNCS 10227, pages 389–403. Springer, 2017.
33. M. Kotoun, P. Peringer, V. Soková, and T. Vojnar. Optimized PredatorHP and the SV-COMP heap and memory safety benchmark (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 942–945. Springer, 2016.
34. D. Kroening and M. Tautschnig. Cbmc: C bounded model checker (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 389–391. Springer, 2014.
35. K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. Residual investigation: predictive and precise bug detection. In *Proc. ISSTA*, pages 298–308. ACM, 2012.
36. R. Majumdar and K. Sen. Hybrid concolic testing. In *Proc. ICSE*, pages 416–426. IEEE, 2007.
37. J. Morse, M. Ramalho, L. C. Cordeiro, D. Nicole, and B. Fischer. Esbmc 1.22 (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 405–407. Springer, 2014.

38. J. Mrázek, M. Jonás, V. Still, H. Lauko, and J. Barnat. Optimizing and caching SMT queries in SymDIVINE (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 390–393. Springer, 2017.
39. P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *Proc. FM*, LNCS 6664, pages 73–87. Springer, 2011.
40. A. Nutz, D. Dietsch, M. M. Mohamed, and A. Podelski. Ultimate Kojak with memory safety checks (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 458–460. Springer, 2015.
41. Z. Rakamarić and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *Proc. CAV*, LNCS 8559, pages 106–113. Springer, 2014.
42. H. Rocha, R. S. Barreto, L. C. Cordeiro, and A. D. Neto. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *Proc. IFM*, LNCS 7321, pages 128–142. Springer, 2012.
43. W. Rocha, H. Rocha, H. Ismail, L. C. Cordeiro, and B. Fischer. DepthK: A k-induction verifier based on invariant inference for C programs (competition contribution). In *Proc. TACAS*, LNCS 10206, pages 360–364. Springer, 2017.
44. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
45. P. Schrammel and D. Kroening. 2LS for program analysis (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 905–907. Springer, 2016.
46. K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. In *Proc. FSE*, pages 263–272. ACM, 2005.
47. P. Shved, M. U. Mandrykin, and V. S. Mutilin. Predicate analysis with Blast 2.7 (competition contribution). In *Proc. TACAS*, LNCS 7214, pages 525–527. Springer, 2012.
48. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. ISSTA*, pages 97–107. ACM, 2004.