





Case Study on Verification-Witness Validators: Where We Are and Where We Go

Dirk Beyer¹  and Jan Strejček² 

¹ LMU Munich, Munich, Germany

² Masaryk University, Brno, Czechia

Abstract. Software-verification tools sometimes produce incorrect answers, which can be a false alarm or a wrong claim of correctness. To increase the reliability of verification results, many verifiers now accompany their answers by witnesses in an interoperable standard format. There exist witness validators that can examine the witnesses and potentially confirm the verification results. This case study analyzes the quality of existing witness validators for C programs using the witnesses produced by a wide variety of 40 verification tools that participated in SV-COMP 2022. In particular, we show that many witness validators sometimes confirm witnesses that are invalid. To remedy this situation, we suggest some advances in witness validation, including a regular comparative evaluation of validators. Our suggestions were recently adopted by the SV-COMP community for the next edition of the competition.

Keywords: Software verification · Program analysis · Software validation · Software bugs · Verification witnesses · Evaluation · Benchmarking

1 Introduction

There are now many tools for verification of computer programs, but as far as we know, none of them claims to always produce correct results. The results of the *Competition on Software Verification (SV-COMP)* show that out of the 57 verifiers participating in the main category called *Overall* in the last five years (there were 10, 13, 11, 10, and 13 participants in this category in years 2018–2022, respectively), only four provide no incorrect results, namely *ULTIMATE KOJAK* in 2018, *CPA-SEQ* and *SYMBIOTIC* in 2019, and *GOBLINT* in 2022. Moreover, communication with industrial developers reveals that even a relatively small portion of incorrect results can devalue credibility of a verification tool. As a solution, many verifiers now accompany their verification results by some evidence in the form of *verification witnesses*. These verification witnesses can be independently analyzed and potentially confirmed by *witness validators*. Industrial developers can use witness validation to triage the verification results: the results with unconfirmed witnesses are ignored and attention is focused on the confirmed ones.

Independent validation of verification witnesses is possible thanks to a machine-readable exchange format for witnesses. The first such format [11] was introduced in 2015. It supported only *violation witnesses* (also called *counterexamples*)

produced when a verifier reports that a given program violates a considered safety specification. The authors of this format also extended the verification tools CPACHECKER and ULTIMATE AUTOMIZER to support validation of these witnesses. In 2016, the format was extended to accommodate also witnesses for the cases when a verifier decides that a given program satisfies a given specification [9]. Such witnesses are called *correctness witnesses*, and they should contain some hints for the proof of program correctness. In the same year, the two mentioned tools were extended to support validation of correctness witnesses as well. In 2018, a new (execution-based) approach for checking of violation witnesses was introduced and implemented in tools CPA-WITNESS2TEST and FSHELL-WITNESS2TEST [12]. Another two witness validators called METAVAL [14] and NITWIT [21] were introduced in 2020, followed by validators DARTGNAN [19] and SYMBIOTIC-WITCH [1] introduced in 2022. The evolution of the witness format and validators is driven by the SV-COMP community. Since SV-COMP 2021, the competition rewards with points only the verification results with witnesses confirmed by at least one witness validator (with the exception of several categories for which witness confirmation is not required for correctness witnesses due to unavailability of suitable witness validators).

The witness format [10,11] is based on GraphML. Each witness contains information about the corresponding verification task (in particular, the program and the specification) and the verification result it witnesses. The main part of the witness resembles an automaton decorated with additional information. Hence, we talk about *witness automata*. A violation witness automaton represents a set of program paths and it is *valid* if at least one of these paths is feasible and violates the considered specification. Figure 1 provides an example of a C program that violates the specification that function `reach_error` is never called, and three different violation witnesses. In general, a violation witness automaton describes a set of program paths by specifying passed program locations (depicted by line numbers on edges), called functions, taken branches, constraints on variable values, etc. Each violation witness automaton has to contain at least one error state representing a specification violation (depicted in red). Further, it can also contain sink states (depicted in blue) saying that the represented paths violating the specification are elsewhere. A witness can represent a single program path by specifying all program inputs (as in Fig. 1b), it can say nothing about input values and prescribe taken branches (as in Fig. 1c), or it can combine some branching information with restrictions on input values (as in Fig. 1d).

A correctness witness automaton represents program invariants and it is *valid* if all these invariants hold and the corresponding program satisfies the considered specification. Ideally, a correctness witness contains a minimal set of invariants implying that the program satisfies the specification. Figure 2 shows a fixed version of the C program (see the rectangle), which can be proven correct, and the correctness witness shows invariants (depicted in green) that help to re-establish the proof of correctness.

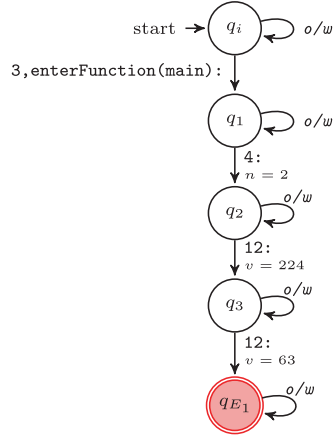
The examples of witnesses are adopted from literature [10] which provides their detailed description: in Sect. 4.2, Examples 7 and 8 explain the violation

```

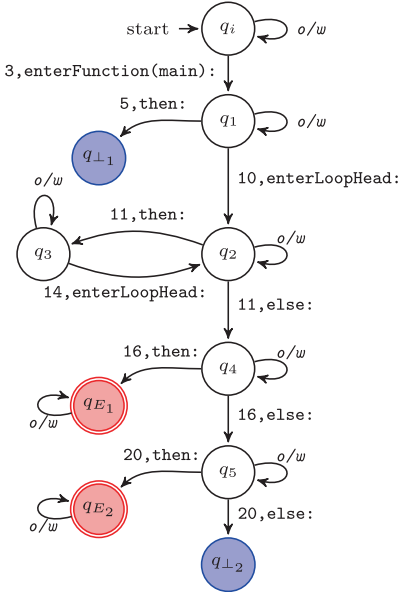
1 void reach_error(){}
2 extern unsigned char
  ↪ __VERIFIER_nondet_uchar(void);
3 int main() {
4   unsigned char n =
  ↪ __VERIFIER_nondet_uchar();
5   if (n == 0) {
6     return 0;
7   }
8   unsigned char v = 0;
9   unsigned char s = 0;
10  unsigned int i = 0;
11  while (i < n) {
12    v = __VERIFIER_nondet_uchar();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

```

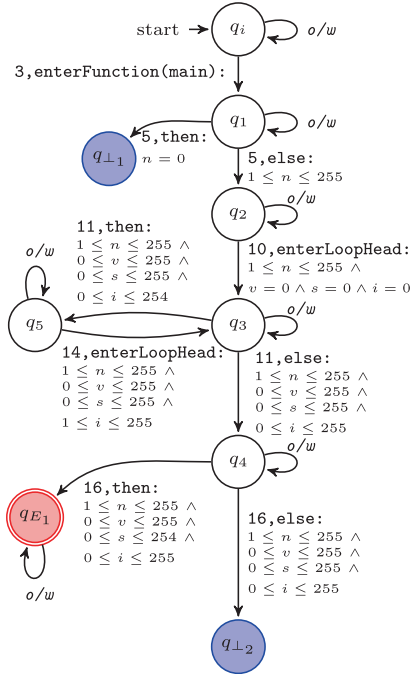
(a) Unsafe program `linear-inequality-inv-b.c`



(b) Violation witness (test values)



(c) Violation witness (branching)



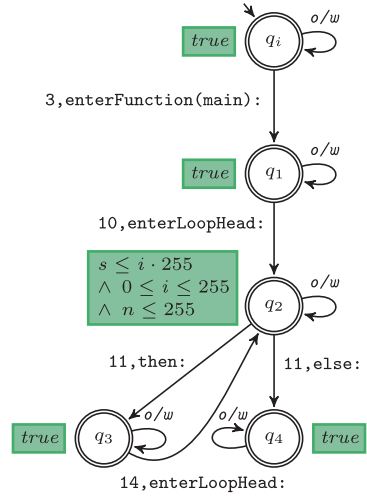
(d) Violation witness (intervals)

Fig. 1. Example C program with a bug (a) and violation witnesses for it: with test values (b), with branching information (c) and with intervals (d); taken from [10]

```

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

```

(a) Safe program `linear-inequality-inv-a.c`

(b) Correctness witness

Fig. 2. Corrected C program (a) and a correctness witness for it (b); the only difference to Fig. 1a is the corrected type in line 9 (highlighted); taken from [10]

witnesses (pages 21–27), and in Sect. 4.3., Example 9 explains the correctness witness (pages 31–33). The witness format admits also *trivial witnesses* that provide no useful information. A trivial violation witness represents all program paths and a trivial correctness witness provides no invariant. Validation of a trivial witness is as hard as the original verification task.

Overview and Outline. A witness validator is given a witness and the corresponding verification task, and it aims at confirming the verification result by proving that the witness is valid.¹ On one side, the addition of the witness-validation step to the verification process increases the reliability of the confirmed verification results. On the other side, the reliability of witness validators is not challenged or even properly studied. As validators are often implemented using the same techniques as their corresponding verifiers (and by the same development teams), it is reasonable to expect that they also sometimes produce incorrect results.

In Sect. 2, we focus on the first goal of this paper, namely to evaluate the performance and reliability of current witness validators for C programs.² There are currently 8 such validators which can be divided into several categories according to their approach.

¹ Note that the current SV-COMP rules use the term *invalid* for witnesses that are not syntactically correct. In our case study, we ignore such witnesses as they can be filtered out by WITNESSLINT (<https://github.com/sosy-lab/sv-witnesses/tree/svcomp22/lint>).

² There are only very few validators that support other languages. We know only about GWIT [18] and WIT4JAVA [20] for Java programs.

- CPACHECKER [11], METAVAl [14], and ULTIMATE AUTOMIZER [11] create a product of a witness automaton and the original program and analyze it. A violation witness is confirmed if the product exhibits the specification violation described by the witness. A correctness witness is confirmed if the product satisfies the specification and the invariants in the witness are valid (cf. [16], Sect. 4.3).
- CPA-WITNESS2TEST [12], CPROVER-WITNESS2TEST (originally called FSHELL-WITNESS2TEST) [12], and NITWIT [21] can handle only violation witnesses. They derive a single test from a given witness automaton and execute it. The witness is confirmed if the execution violates the considered specification.
- SYMBIOTIC-WITCH [1] can process also only violation witnesses. It performs symbolic execution of the given program and tracks the corresponding set of states in the witness automaton. A witness is confirmed if the symbolic execution violates the considered specification and the tracked set contains an error state of the witness automaton.
- DARTGNAN [19] is a bounded model checker for parallel programs, which has been extended with the ability to analyze violation witnesses. It transforms the violation witness and the program into an SMT query, and it confirms the witness if the query is satisfiable.

We evaluate the validators on witnesses produced in SV-COMP 2022. As various validators support different specifications and program features, they are applicable only to witnesses created for verification tasks of selected SV-COMP categories. Verification tasks with C programs are currently divided into 6 main categories, which can be roughly characterized as follows.

- *ReachSafety* contains sequential programs that should be checked for unreachability of a given error function.
- *MemSafety* consists of sequential programs that should be checked to contain no invalid dereference, no invalid deallocation, and no memory leaks.
- *ConcurrencySafety* contains parallel programs that should be checked for unreachability of a given error function.
- *NoOverflows* collects sequential programs that should contain no overflow of a signed integer.
- *Termination* consists of sequential programs that should be checked to have no infinite execution.
- *SoftwareSystems* collects more complex programs that are usually a part of real software projects and they should be checked for specifications described in *ReachSafety*, *MemSafety*, or *NoOverflows*.

The applicability of the considered validators to violation and correctness witnesses of individual SV-COMP categories is summarized in Table 1. Please note that even if the table indicates that a certain validator is applicable to violation or correctness witnesses of a certain category, it does not mean that the validator can handle all such witnesses of this category (for example, a validator may not support a specific feature of some programs).

Verification tasks in SV-COMP are labelled with expected verification results. We consider the labelling with expected results as highly reliable due

Table 1. Applicability of validators to violation and correctness witnesses from individual SV-COMP categories; some validator names are abbreviated

Category	violation-witness validators							correctness-witness validators			
	CPACHECKER [11]	CPA-W2T [12]	CPROVER-W2T [12]	DARTGNAN [19]	METAVAL [14]	NITWIT [21]	UATOMIZER [11]	SYMBIOTIC-WITCH [1]	CPACHECKER [9]	METAVAL [14]	UATOMIZER [9]
ReachSafety	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
MemSafety	✓	✓	✓		✓		✓	✓		✓	✓
ConcurrencySafety	✓			✓							
NoOverflows	✓	✓	✓		✓		✓	✓	✓	✓	✓
Termination	✓				✓		✓				
SoftwareSystems	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓

to the following penalty mechanism of SV-COMP and competitiveness of its community. In SV-COMP, if a verifier produces an incorrect result (i.e., the opposite to the expected one), it immediately gets many penalty points. If the authors of the verifier are confident that the result is correct, they can (and often do) challenge the expected result. The verification task is then discussed and potentially relabelled.³ Unfortunately, there is no set of witnesses labelled as valid or invalid, and we cannot safely assume that all witnesses accompanying correct verification results are valid. In fact, there are known cases of correct verification results accompanied by invalid witnesses. For example, this is the case of some violation witnesses produced by SYMBIOTIC 9 for some *MemSafety* benchmarks [17]. However, when a verifier produces an incorrect verification result, the corresponding witness has to be invalid. In our experiments, we apply the existing witness validators on all relevant witnesses of both correct and incorrect verification results computed in SV-COMP 2022.

Section 3 is devoted to the second goal of this paper: to initiate qualitative improvement of witness validators. In particular, we suggest extending the semantics of possible validator outcomes and we propose a formula for evaluating validators. Our suggestions have been recently accepted by the SV-COMP community and a new competition track for witness validators has been announced starting from SV-COMP 2023.

Related Work. Existing papers on witness validators typically present only the confirmation rates of considered validators on the set of witnesses accompanying correct verification results, which are implicitly assumed to be valid

³ For example, see [Merge Request 1336](#) of the [SV-Benchmarks repository](#).

witnesses [10, 14, 19, 21]. Evaluation of validators on invalid witnesses accompanying incorrect verification results has been previously done only twice: in 2015 for a limited set of invalid violation witnesses and the initial versions of witness validators CPACHECKER and ULTIMATE AUTOMIZER [11] and in 2018 for a larger set of invalid violation witnesses and initial versions of witness validators CPA-WITNESS2TEST and CPROVER-WITNESS2TEST and then-current versions of CPACHECKER and ULTIMATE AUTOMIZER [12]. In contrast, we consider invalid verification witnesses for both violation and correctness results and all 8 currently available witness validators in their versions used in SV-COMP 2022.

More information about witnesses and their validation in the context of SV-COMP can be found in regular competition reports [5, 6]. There is also a study [4] on violation and correctness witnesses produced in SV-COMP 2019.

2 Evaluation

We would like to investigate the state of the art of witness validation. Therefore, we take a large set of 158 848 known syntactically correct witnesses from SV-COMP 2022 and validate all those witnesses using all available witness validators for C programs and report the results.

Execution Environment. We executed all experiments on a cluster with 167 machines, each with a CPU of type Intel Xeon E3-1230 v5, 3.4 GHz, with 8 processing units (virtual cores), 33 GB RAM, operating system Ubuntu 20.04 (Linux 5.4.0-94-generic). Each validation run (execution of one validator on one verification task and witness) was limited to 2 processing units, 7 GB memory, and 900 s of CPU time for correctness validators and 90 s of CPU time for violation validators. We chose this configuration because it was used in SV-COMP 2022. In order to ensure reliable measurement and control of the computing resources and isolation of processes, we used the benchmarking framework BENCHEXEC [13].

Evaluated Validators. In this evaluation, we consider all eight witness validators for C programs that participated in SV-COMP 2022. Table 1 lists the validators and the categories for which they can validate witnesses.

Data Set and Benchmark. The witnesses and the verification tasks (program and specification) are taken from the data set of SV-COMP 2022 at Zenodo [8]. SV-COMP organizes the verification tasks with C programs into six categories. We take all witnesses produced for these tasks by all participating verification tools. Then we remove the witnesses for which WITNESSLINT produced an exception. Exceptions are typically caused by syntax problems or too large witness files.

We classify each violation witness for a correct program as invalid (because the competition classified the result of the verifier as false alarm), and we classify each correctness witness for a buggy program as invalid (because the competition classified the result of the verifier as wrong claim of correctness). All other witnesses are classified as valid*, because they do not contradict the expected result. We use the term valid* with asterisk because there are witnesses that do not contradict the expected result but are still invalid (e.g., there can be a violation witness

Table 2. Validation of violation witnesses by eight violation validators; the numbers are hyperlinked to the tables generated by BENCHEXEC

Category	Witnesses	CPACHECKER	CPA-w2T	CPROVER-w2T	DARTGNAN	METAVAL	NITWIT	SYMBIOTIC -WITCH	UAUTOMIZER
ReachSafety									
valid*	26 797	14 908	8628	14 168	–	0	15 507	11 176	8592
invalid	5177	28	12	2	–	0	10	0	0
MemSafety									
valid*	16 984	12 594	231	954	–	116	–	8394	4197
invalid	2804	0	0	26	–	2	–	0	0
ConcurrencySafety									
valid*	4746	2700	–	–	1464	–	–	–	–
invalid	1293	40	–	–	0	–	–	–	–
NoOverflows									
valid*	2808	2334	887	1436	–	1982	–	2609	2468
invalid	167	0	0	0	–	0	–	0	0
Termination									
valid*	3652	2580	–	–	–	598	–	–	960
invalid	56	21	–	–	–	0	–	–	0
SoftwareSystems									
valid*	2102	621	6	33	–	0	0	179	26
invalid	5903	5	0	27	–	0	0	51	4

representing no feasible path violating the considered specification, even if such a path exists). However, there is currently no reliable way to automatically identify invalid witnesses that do not contradict the expected result. Tables 2 and 3 report in column ‘Witnesses’ the number of valid* and invalid witnesses for each category.

Results. We report the results of our validation experiments in two tables. The results on violation witnesses are presented in Table 2 and the results on correctness witnesses in Table 3. For each category and validator, row ‘valid*’ reports the number of valid* witnesses confirmed by the validator, and row ‘invalid’ reports the number of invalid witnesses erroneously confirmed by the validator. Due to the source of invalid witnesses described above, each erroneous confirmation of an invalid witness here means that the validator either confirmed a violation witness, but the program does not violate the specification, or it confirmed a correctness witness, but the program does violate the specification. In the following we highlight a few observations revealed by the results.

Table 3. Validation of correctness witnesses by three correctness validators; the numbers are hyperlinked to the tables generated by BENCHEXEC

Category	Witnesses	CPACHECKER	METAVAL	UAUTOMIZER
ReachSafety				
valid*	31 013	17 312	19 655	19 632
invalid	894	0	315	3
MemSafety				
valid*	16 948	–	227	14 384
invalid	326	–	0	0
ConcurrencySafety				
valid*	3177	–	–	–
invalid	389	–	–	–
NoOverflows				
valid*	2089	1718	1608	1713
invalid	300	0	36	0
Termination				
valid*	4502	–	–	–
invalid	14	–	–	–
SoftwareSystems				
valid*	25 819	6771	20 624	19 343
invalid	888	0	403	0

Soundness of validators. There is only one validator, namely DARTGNAN, that does not confirm any invalid violation witness. The validator participated only in category *ConcurrencySafety* as it is specialized in parallel programs (Table 2). CPACHECKER does not confirm any invalid correctness witness (Table 3).

There seems to be a particularly difficult category. The category **SoftwareSystems** has a large number of invalid violation witnesses (Table 2, ‘Witnesses’ column). This means that in this category, many verification runs report a false alarm for a correct program, accompanied by an invalid violation witness. The violation witnesses in this category seem to be difficult for validation, as only CPACHECKER confirmed more than 10 % of valid* violation witnesses. Moreover, all validators that confirmed at least ten valid* violation witnesses confirmed also some invalid violation witnesses.

Our evaluation revealed technical problems. The validator METAVAL does not confirm any violation witness (Table 2) in categories **ReachSafety** and **SoftwareSystems** and confirms a large number of invalid correctness witnesses (Table 3) in these categories. The reason for those incorrect validation results is that the validator was not adapted to a new rule of SV-COMP that was introduced for SV-COMP 2021: All verification tasks in those categories were changed to using a new logic to encode invalid function calls. Other specifications are not affected by this change.

Summary. Most of the invalid witnesses that were incorrectly confirmed were due to bugs in validators. The conclusion is that the quality of validators should be increased by establishing means to stimulate the inspection and quality control of validation tools. A competition track for validators suggested in the following section could help drawing the attention of developers to inspecting results of validators. Currently, SV-COMP uses validators for confirmation of verification results, but does not evaluate the quality of their results.

Threats to Validity. Regarding internal validity, the main threat to our results is that we rely on the expected results for verification tasks. If those were incorrectly specified, our classification of validator results would also be incorrect. But the verification tasks in the benchmark collection that we use are actively maintained by the community and the participating teams inspected the results of their verifiers. The 33 actively participating teams in SV-COMP 2022 have approved the results of their verifiers before the results were published.

For executing the experiments, we used the publicly-available benchmarking framework `BENCHEXEC` [13], which gives us access to the modern features of the Linux kernel for controlling the resources and for isolating executions. This framework is used by several competitions and is actively maintained. For job distribution on the cluster we use `VERIFIERCLOUD`, which is also used by several competitions and research groups for their lab work. It is unlikely that a bug in the benchmarking infrastructure causes wrong results.

Regarding external validity, our results are specific to witness validators for the programming language C, because this is the only language for which a large set of verification and validation tools exist. The first two validators [18, 20] for Java were introduced for SV-COMP 2022. Further, our results are specific to validators that participated in SV-COMP and to the verification tasks from the SV-Benchmarks collection. We are not aware of any validators besides those participating in the competition, and we are not aware of a benchmark that is better suited for the evaluation than what is used by the competition. Therefore, we assume that our results are still significant because SV-COMP is comprehensive.

3 Suggestions for Advances in Witness Validation

Extended Semantics of Validator Outcomes. Possible validator answers recognized by SV-COMP are the same as possible answers of verifiers, which are

- **false**, meaning that the given program violates the given specification and a violation witness was generated,
- **true**, meaning that the given program satisfies the given specification and a correctness witness was generated, and
- **unknown**, meaning that the verifier was unable to decide.

The interpretation of a witness-validator answer depends on the kind of the analyzed witness. A violation witness is confirmed if a validator outputs **false**. All other answers (including **true** and **unknown**) mean that the witness is not confirmed by this validator. Similarly, a correctness witness is confirmed if a validator

outputs `true` and all other answers mean that the validator did not confirm the witness. In other words, even if a validator has the confidence to say that some witness is invalid, the competition rules give it the same semantics as `unknown`. As a consequence, there is no difference between witnesses that are not confirmed due to insufficient power of validators and those that were refuted by some validators.

We suggest to explicitly state the semantics of a validator output as follows. On violation witnesses, a validator should produce

- `false` to confirm that there exists a program execution represented by the witness such that it violates the considered specification,
- `true` to refute the witness as there is no program execution represented by the witness that violates the considered specification, or
- `unknown` to indicate that it is unable to decide.

On correctness witnesses, a validator produces

- `false` to refute the witness as there exists some execution violating the considered specification or some invariant given in the witness,
- `true` to confirm the witness as the validator can prove that the program satisfies the considered specification with help of the invariants given in the witness and that all invariants given in the witness are valid, or
- `unknown` to indicate that it is unable to decide.

Evaluation of Validators. One can find many areas of computer science (e.g., SMT solving), where some kind of competition or regular evaluation led to a rapid improvement of the state of the art. With this motivation, we suggest to extend SV-COMP with a comparative evaluation of witness validators, and we propose the following scoring schema for this evaluation.

Assume that we are given a witness validator, a set of `valid*` witnesses, and a set of invalid witnesses. Our scoring schema is inspired by the established scoring schema for evaluating verifiers in SV-COMP. The community agreed that showing that a system satisfies a given specification deserves more credit than showing that the specification is violated. Hence, SV-COMP rewards correct (and confirmed) answers `true` with 2 points and correct (and confirmed) answers `false` with 1 point. The penalty factor for incorrect answers is -16 , which means that incorrect `true` yields -32 points and incorrect `false` -16 points.

The proposed scoring schema for validators is depicted in Fig. 3. We first describe the scores for invalid violation witnesses (the right side of the figure). Refutation of an invalid witness is rewarded with 2 points as it means to decide that all program paths represented by the witness satisfy the specification, which is an analogy to showing that a program satisfies its specification. Refutation of an invalid correctness witness is rewarded with 1 point as it corresponds to finding a violation of the specification or some invariant given in the witness. Confirmation of an invalid witness yields the penalty p for a violation witness and $2p$ for a correctness witness, where p is the *penalty factor* (with $p < 0$). Points and penalties for invalid witnesses are accumulated in p_{invalid} . The proposed scores for `valid*` witnesses (the left side of the figure) reflect the fact that these witnesses

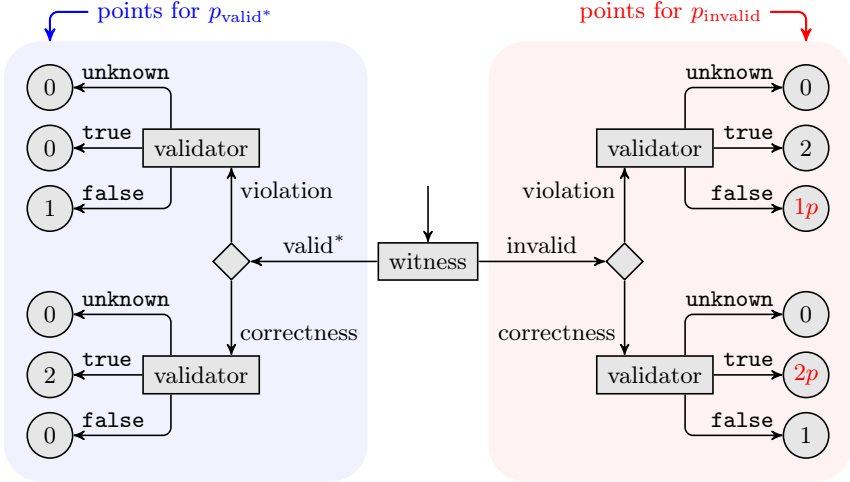


Fig. 3. Proposed scoring schema for evaluation of validators (with $p < 0$)

are only assumed to be valid and some of them can be actually invalid. Hence, we suggest to reward only confirmation of valid^* witnesses: 2 points for each confirmed correctness witness and 1 point for each confirmed violation witness. Points for valid^* witnesses are accumulated in p_{valid^*} .

One can observe in Tables 2 and 3 that the number of incorrect witnesses is typically one or two orders of magnitude lower than the number of valid^* witnesses and this disbalance is assumed to increase if verifiers produce less incorrect verification results. Further, the p_{invalid} deserves a higher impact than p_{valid^*} as we do not really know whether valid^* witnesses are indeed valid. Hence, we propose to compute the score as the sum

$$\text{score} = \frac{p_{\text{valid}^*}}{|\text{valid}^*|} + q \cdot \frac{p_{\text{invalid}}}{|\text{invalid}|}$$

where the points in p_{valid^*} and p_{invalid} are normalized by the cardinality of the corresponding witness sets and p_{invalid} is given a higher weight using the factor q .

We suggest to compute the validator scores separately for witnesses of each category. The overall score of a validator can be computed by the normalization used in SV-COMP to compute the overall scores of verifiers (see [3], page 597).

Our proposal of a comparative evaluation of witness validators based on the scoring schema above was presented and discussed at the SV-COMP community meeting on April 7, 2022. The community decided to establish a witness-validation track from SV-COMP 2023 onwards. The community further decided to use the suggested scoring schema and set the parameters to $p = -16$ and $q = 2$.

4 Conclusion

Verification tools are complicated software systems, which naturally contain conceptual and programming mistakes. Therefore, it is imperative to apply validators to ensure that a verification engineer is not bothered with incorrect verification results. Our case study investigates the correctness of witness validators, in particular, how many invalid witnesses are confirmed by validators. The results indicate that there is room for improvement of the validators. We initiated the extension of SV-COMP by a comparative evaluation of witness validators that will utilize the full set of validator answers and use the presented scoring schema for ranking validators. If there is an incentive, then there will be improvement, as is shown by the enormous success of competitions in the field of formal methods [2].

Data-Availability Statement. Our experiments are based on publicly available data sets from SV-COMP 2022, where a large number of verification tasks [7] was executed and a large number of verification witnesses [8] was produced. The witness format is maintained in a GitHub repository: <https://github.com/sosy-lab/sv-witnesses/tree/svcomp22/>. Our experimental results are available on a supplementary web page (<https://sv-comp.sosy-lab.org/2022/results/validators/>) as tables produced by BENCHEXEC [13] (also linked to from Tables 2 and 3). The log output is available by clicking on the status of a result in the tables. All experimental results (raw data, tables) and scripts are available in our reproduction package [15].

References

1. Ayaziová, P., Chalupa, M., Strejček, J.: SYMBIOTIC-WITCH: A Klee-based violation witness checker (competition contribution). In: Proc. TACAS (2). pp. 468–473. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_33
2. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS (3). pp. 3–24. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_1
3. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
4. Beyer, D.: A data set of program invariants and error paths. In: Proc. MSR. pp. 111–115. IEEE (2019). <https://doi.org/10.1109/MSR.2019.00026>
5. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24
6. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
7. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). <https://doi.org/10.5281/zenodo.5831003>

8. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). <https://doi.org/10.5281/zenodo.5838498>
9. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). <https://doi.org/10.1145/2950290.2950351>
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). <https://doi.org/10.1145/3477579>
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). <https://doi.org/10.1145/2786805.2786867>
12. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
13. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). <https://doi.org/10.1007/s10009-017-0469-y>
14. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
15. Beyer, D., Strejček, J.: Reproduction package for article ‘case study on verification-witness validators: Where we are and where we go’. Zenodo (2022). <https://doi.org/10.5281/zenodo.7096382>
16. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
17. Chalupa, M., Řečtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: SYMBIOTIC 9: String analysis and backward symbolic execution with loop folding (competition contribution). In: Proc. TACAS (2). pp. 462–467. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32
18. Howar, F., Mues, M.: GWIT (competition contribution). In: Proc. TACAS (2). pp. 446–450. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_29
19. Ponce-De-Leon, H., Haas, T., Meyer, R.: DARTAGNAN: Smt-based violation witness validation (competition contribution). In: Proc. TACAS (2). pp. 418–423. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_29
20. Wu, T., Schrammel, P., Cordeiro, L.: WIT4JAVA: A violation-witness validator for Java verifiers (competition contribution). In: Proc. TACAS (2). pp. 484–489. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_36
21. J. Švejda, Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NITWIT. In: Proc. TACAS. pp. 40–57. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_3

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

