Software Verification Witnesses 2.0

Paulína Ayaziová [1], Dirk Beyer [2]✉, Marian Lingsch-Rosenfeld [2], Martin Spiessl [2], and Jan Strejček [1]

[1] Masaryk University, Brno, Czech Republic
{xayaziov,strejcek}@fi.muni.cz
[2] LMU Munich, Munich, Germany
{dirk.beyer,marian.lingsch,spiessl}@sosy.ifi.lmu.de

Abstract. Verification witnesses are now widely accepted objects used not only to confirm or refute verification results, but also for general exchange of information among various tools for program verification. The original format for witnesses is based on GraphML, and it has some known issues including a semantics based on control-flow automata, limited tool support of some format features, and a large size of witness files. This paper presents version 2.0 of the witness format, which is based on YAML and overcomes the above-mentioned issues. We describe the new format, provide an experimental comparison of various aspects of the original and the new witness format showing that both witness formats perform similarly, and report on its adoption in the community.

Keywords: Verification Witness · Software Verification · Validation · Exchange Format · Invariant · Counterexample

1 Introduction

Software verification is a process that detects bugs in computer programs or proves their absence. Unfortunately, software verifiers can also contain bugs and their verdicts can thus be incorrect. To increase the reliability of the verification process, starting eight years ago, software verifiers have accompanied their verification results with witnesses that justify the verdict and can be independently analyzed by witness validators developed by various teams and based on different techniques.

The first generic format for witnesses of verification results [1] 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. 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 [2]. Such witnesses are called *correctness witnesses*, and they should contain invariants that help to prove that the program is correct. The format was soon adopted by the verification community and by the *Competition on Software Verification (SV-COMP)* [3], which led to fast adoption of the format by many verification tools and to the development of numerous witness validators.

T. Neele and A. Wijs (Eds.): SPIN 2024, LNCS 14624, pp. 184–203, 2025.
https://doi.org/10.1007/978-3-031-66149-5_11

The overview of existing validators can be found in a recent survey [4]. Since 2023, SV-COMP has a new track on witness validation [5].

While the format was originally intended for validation of verification results, some witness validators can also refute a witness [4, 5]. The format soon found also some applications that were not intended at the time of its development. In particular, it is used to exchange information between different verifiers in the context of cooperative verification [6, 7], as a way to provide feedback to a software developer [8, 9], or as a way to combine automatic and interactive verifiers [10]. In 2022, the authors of the format published a paper [11] with its detailed description and with an extensive experimental study on its applications.

Despite the indisputable success of the format, it has also some weaknesses. The format is based on GraphML [12] and witnesses have the form of automata, which makes them easy to visualize, but also lengthy and unsuitable for reading in their textual form. More importantly, the semantics of the format is formally defined over programs represented by *control-flow automata (CFA)*. Unfortunately, there is no standardized translation of programs written in common programming languages like C or Java to CFA. As a result, the semantics of the format over programs in standard languages has some ambiguities. The SV-COMP community even found a part of the semantics related to implicit loop edges as inappropriate and decided to change it. Another issue of the original witness format is connected to the high number of features it provides. For example, if an invariant or an assumption uses variables that appear in different functions or scopes, the format allows to specify the scope for their interpretation. Another example is that the location of some witness event can be specified very loosely by an interval of lines. Practical experience shows that some of these features are not used in any witness generated by verifiers and, what is more alarming, unsupported or even ignored by witness validators. In fact, there is probably no witness validator fully implementing the format. This can lead to the situation in which a valid verification witness employing some less frequent feature is not confirmed or even refuted, or an invalid witness is confirmed.

This paper presents a new generation of the witness format that avoids the mentioned weaknesses. In particular, we use a concise format that is based on YAML, which makes the witnesses shorter in general. Further, the format provides only features that were really used by verification witnesses in the original format. As the format is significantly simpler, it is easy to fully support it by validators. Finally, the semantics of the format is formulated over programming languages using terms and concepts from their standards.

The format itself is described in Sect. 2 and referred to as version 2.0. In its current state, the format supports only sequential programs written in C and basic program properties, namely unreachability of a given error function, unreachability of signed integer overflow, and unreachability of invalid pointer dereference and deallocation. We have adopted two verification tools, namely CPAchecker [13] and Symbiotic [14], to produce verification witnesses in the new format. We have also developed two witness validators, namely CPAchecker [11] (as an extension of the exiting tool) and Witch3 (new validator based on the

concept of Witch3 [15]) to validate witnesses in this format. Using these tools and other tools from the competition, Sect. 3 evaluates the impact of the new format on the witnesses and their validation. Sect. 4 summarizes the differences between the original and the new format and shows the current adoption of the new format by verification and witness validation tools.

Contributions. In this paper, we contribute:

- a new generation of the format for verification witnesses that solves most problems that were present in the previous format,
- a preliminary evaluation of the impact of the new format on the effectivity and performance of witness validation, and
- an overview of a few measures that characterize the new witnesses.

Related Work. Our work certainly stands on the shoulders of the original format for verification witnesses [1, 2], but we claim to provide a substantial improvement over the original format by addressing its weaknesses (see Sect. 4). Witnesses are used ubiquitously in areas where algorithms have a high computational complexity. For example, witnesses are used for certifying graph algorithms [16]. Turing used assertions [17] already and argued that one should justify the correctness of programs. In the area of logic solvers, witnesses for the results are of essence for competitions, and important competitions require witnesses and their validation. For example, the *Termination Competition (termCOMP)* [18] uses the format CPF [19], the competition of SAT solvers [20] uses the DRAT format [21] together with the validator DRAT-trim [22], and the competition on SMT solving verifies models with Dolmen [23].

Witnesses are not only important to certify the correctness of a solver's answer, it is also important for the goal of explainability: The *true* / *false* answers alone are not as valuable compared to also providing the reasons to understand the answer. For example, witnesses can be used to derive test cases [9] and to aid debugging with visualizations [8]. Execution reports [24] help organize the analysis results, and the format SARIF [25] is used by static analyzers to represent results.

2 Witness Format 2.0

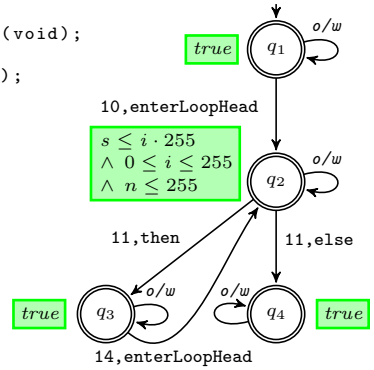
The witness format 2.0 is an extension of the YAML format, version 1.2. Individual verification witnesses are represented by *entries*. Each entry has three key-value pairs. The key `entry_type` has the value `invariant_set` or `violation_sequence` corresponding to the type of the witness: a correctness witness is represented by one or more entries of type `invariant_set`, while a violation witness is represented by a single entry of type `violation_sequence`. Further, the key `metadata` refers to a mapping that describes mainly the context of the witness: the format version used by the entry, the unique identifier of the entry, the creation time of the entry, the tool that produced the entry, and the verification task the witness relates to. Finally, the value of the key `content` represents the semantical content

Table 1: Structure of entries common for violation and correctness witnesses; some nodes are nested; optional items are marked with *; the term *scalar* in YAML refers also to strings

Key	Value	Description
`entry_type`	`invariant_set`	the entry type of a correctness witness
	`violation_sequence`	the entry type of a violation witness
`metadata`	mapping	the context of the witness; see below
`content`	sequence	the witness content; see Tables 2 and 3
content of `metadata`		
`format_version`	`2.0`	the used version of the format
`uuid`	scalar	a unique identifier of the entry; it uses the UUID format defined in RFC4122
`creation_time`	scalar	the date and time of the entry creation; it uses the format given by ISO 8601
`producer`	mapping	the tool that produced the entry; see below
`task`	mapping	the verification task to which the entry is related; see below
content of `producer`		
`name`	scalar	the name of the tool
`version`	scalar	the version of the tool
`configuration`*	scalar	the configuration in which the tool ran
`command_line`*	scalar	the command line with which the tool ran; it should be a bash-compliant command
`description`*	scalar	any additional information
content of `task`		
`input_files`	sequence	the list of files given as input to the verifier, e.g. `["path/f1.c", "path/f2.c"]`
`input_file_hashes`	mapping	SHA-256 hashes of all files in `input_files`, e.g. `{"path/f1.c": 511..., "path/f2.c": f70...}`
`specification`	scalar	the property considered by the verifier; it uses the SV-COMP format given at https://sv-comp.sosy-lab.org/2024/rules.php
`data_model`	`ILP32` or `LP64`	the data model considered for the task
`language`	`C`	the programming language of the input files; the format currently supports only C

of the entry. The key-value pairs are presented in a structured way in Table 1. The table also presents the key-value pairs of the nested mapping `metadata` and its nested mappings `producer` and `task`. We describe the possible values of the

```
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 int  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 }
```



Specification:
`G ! call(reach_error())`,
i.e., all calls of `reach_error()`
are unreachable

```
1 - entry_type: invariant_set
2   metadata: <...>
3   content:
4     - invariant:
5         type: loop_invariant
6         location:
7           file_name: "inv-a.c"
8           line: 11
9           column: 1
10          function: main
11        value: "s <= i*255 && 0 <=
          ↪ i && i <= 255 && n <= 255"
12        format: c_expression
```

Fig. 1: Example C program inv-a.c taken from [11] (top left) satisfying the given specification (bottom left) and equivalent correctness witnesses in format 1.0 (top right, visualized as automaton) and format 2.0 (bottom right), with a single nontrivial invariant

key `content` in the following subsections separately for correctness witnesses and violation witnesses as they are conceptually different.

2.1 Correctness Witnesses

Correctness witnesses provide invariants that should help to prove the program correct. In the old format (1.0), invariants are tied to automata nodes and these nodes can correspond to multiple program locations and various moments of program executions. The new format (2.0) simply assigns invariants to program locations. Figure 1 provides an example of a correctness witness in the old format and in the new format.

Syntax. In entries of type `invariant_set` which represent a correctness witness, the key `content` contains a sequence of zero or more *invariants*. An `invariant` is a mapping with the following four keys.

`type` has the value `loop_invariant` if the invariant is assigned to a loop head and the value `location_invariant` if it is assigned to another location.

Table 2: Structure of the `content` part of entries representing correctness witnesses; optional items are marked with *

Key	Value	Description
`content`	sequence	a sequence of one or more `invariant` elements
description of `invariant`		
`invariant`	mapping	a basic building block of correctness witnesses; see below
content of `invariant`		
`type`	`loop_invariant`	the invariant type for iteration statements
	`location_invariant`	the invariant type for arbitrary statements
`location`	mapping	the location of the invariant; see below
`format`	`c_expression`	the invariant is a C expression
`value`	scalar	the actual invariant
content of `location`		
`file_name`	scalar	the file of the location
`line`	scalar	the line number of the location
`column`*	scalar	the column of the location
`function`*	scalar	the name of the function containing the location

`location` of a `loop_invariant` must point to the first character of a keyword at the beginning of a loop (i.e., `for`, `while`, or `do`). The `location` of a `location_invariant` must point to the first character of a statement or a declaration that is within a compound statement.

`format` has the value `c_expression` as the format currently supports only invariants that are C expressions.

`value` holds the actual invariant string (e.g., `"s <= i*255 && i > 0"`), which is a side-effect-free C expression over variables in the scope where the invariant is placed.

The `location` is a mapping with mandatory keys `file_name` that holds the name of the file and `line` representing the line number (the first line has the number 1). Additionally, there are two optional keys called `column` and `function`. The key `column` specifies the column number of the location (value 1 is the position of the first character on the `line`). If the column is not given, then it is interpreted as the leftmost suitable position on the line, where suitability is given by `type` and the restrictions given above. The key `function` provides the name of the function containing the location. Technically, this information is superfluous as it is determined by the `file_name`, `line`, and `column`. It is therefore not intended for any algorithmic processing of the witness, but only to improve human readability of the witness.

The structure of `content` and its nested items are summarized in Table 2.

Semantics. The correctness witness is *valid* if it fulfills the following requirements.

- Each `loop_invariant` must always hold immediately before evaluating the condition of the corresponding loop.
- Each `location_invariant` must always hold immediately before evaluating the corresponding statement or declaration.
- The specification must be satisfied for all program executions.
- No invariant evaluation causes undefined behavior and no undefined behavior occurs during any execution of the program.

Note that the order of invariants in an `invariant_set` or their division into several entries of type `invariant_set` is not important. The semantics also reveals the difference between the two types of invariants: if we replace `loop_invariant` with `location_invariant`, then the invariant has to hold only before the loop is executed, but not after each loop iteration.

2.2 Violation Witnesses

A violation witness should describe a program execution violating the considered property. For brevity, the violating execution is described loosely and the witness thus represents a set of such executions. In the old format (1.0), a violation witness is an automaton with edges prescribing consecutive restrictions on program executions. The automaton can contain various branches and loops. In the new format (2.0), a violation witness is a sequence of *waypoints* that have to be passed by the executions. To make the witness validation more efficient, the format also allows specifying waypoints that have to be avoided. Figure 2 provides an example of a violation witness in the old format and in the new format.

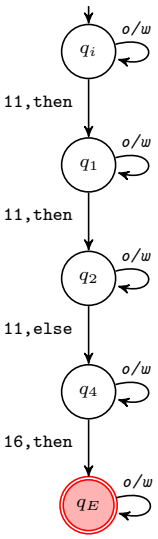
Syntax. The basic building blocks of violation witnesses in the new format are waypoints. Technically, a `waypoint` is a mapping with four keys, namely `type`, `location`, `constraint`, and `action`. The values of the first three keys specify a requirement on a program execution to pass a waypoint: `type` describes the type of the requirement, `location` ties the requirement to some program location, and `constraint` gives the requirement itself. The key `action` then states whether the executions represented by the witness should pass through the waypoint (value `follow`) or avoid it (value `avoid`). The format currently supports five possible values of `type` with the following meanings:

`assumption` The `location` has to point to the first character of a statement or a declaration within a compound statement. A requirement of this type says that a given constraint holds before evaluating the pointed statement or declaration. The `constraint` is a mapping with two keys: `format` specifies the language of the assumption and `value` contains a side-effect-free assumption over variables in the current scope. The value of `format` is `c_expression` as C expressions are the only assumptions currently supported. In the future, we plan to support also assumptions in ACSL [26].

`branching` A requirement of this type says that a given branching is evaluated in a given way. The `location` points to the first character of a branching keyword like `if`, `while`, `switch`, or to the character `?` in the ternary operator (`?:`).

```c
void reach_error(){}
extern unsigned char
 ↪ __VERIFIER_nondet_uchar(void);
int main() {
  unsigned char n = __VERIFIER_nondet_uchar();
  if (n == 0) {
    return 0;
  }
  unsigned char v = 0;
  unsigned char s = 0;
  unsigned int  i = 0;
  while (i < n) {
    v = __VERIFIER_nondet_uchar();
    s += v;
    ++i;
  }
  if (s < v) {
    reach_error();
    return 1;
  }
  if (s > 65025) {
    reach_error();
    return 1;
  }
  return 0;
}
```

Specification:
`G ! call(reach_error())`,
i.e., all calls of `reach_error()`
are unreachable




```yaml
- entry_type:
 ↪ violation_sequence
  metadata: <...>
  content:
    - segment:
      - waypoint:
          action: follow
          type: branching
          location:
            file_name:
             ↪ "inv-b.c"
            line: 11
          constraint:
            value: true
    - segment:
      - waypoint:
          action: follow
          type: branching
          location:
            file_name:
             ↪ "inv-b.c"
            line: 11
          constraint:
            value: true
    - segment:
      - waypoint:
          action: follow
          type: branching
          location:
            file_name:
             ↪ "inv-b.c"
            line: 11
          constraint:
            value: false
    - segment:
      - waypoint:
          action: follow
          type: target
          location:
            file_name:
             ↪ "inv-b.c"
            line: 17
```

Fig. 2: Example C program inv-b.c taken from [11] (top left) violating the given specification (bottom left) and similar violation witnesses in format 1.0 (middle, visualized as automaton) and format 2.0 (right)

The `constraint` is then a mapping with only one key `value`. For binary branchings, `value` can be either `true` or `false` saying whether the true branch is used or not. For the keyword `switch`, `value` can be an integer constant or `default`. The integer constant specifies the value of the controlling expression of the `switch` statement. The value `default` says that the value of this expression does not match any case of the `switch` with the exception of the `default` case (if it is present).

`function_enter` The `location` points to the right parenthesis after the function arguments of a function call. The requirement says that the called function is entered. The key `constraint` has to be omitted in this case.

`function_return` Such a requirement says that a given function call has been evaluated and the returned value satisfies a given constraint. The `location` points to the right parenthesis after the function arguments at the function call. The `constraint` is a mapping with keys `format` and `value`. We currently support only ACSL expressions of the form `\result <op> <const_expression>`, where `<op>` is one of `==`, `!=`, `<=`, `<`, `>`, `>=` and `<const_expression>` is a constant expression. The value of `format` has to be `acsl_expression`.

`target` This type of requirement can be used only with action `follow` and it marks the program location where the property is violated. More precisely, the `location` points at the first character of the statement or full expression whose evaluation is sequenced directly before the violation occurs, i.e., there is no other evaluation sequenced before the violation and after the sequence point associated with the `location`. This also implies that it can point to a function call only if it calls a function of the C standard library that violates the property or if the function call itself is the property violation. The key `constraint` has to be omitted.

Waypoints are organized into *segments*. Each `segment` is a sequence of zero or more waypoints with action `avoid` and exactly one waypoint with action `follow` at the end. A segment is called *final* if it ends with the `target` waypoint and it is called *normal* otherwise.

Finally, we can describe the `content` part of `violation_sequence` entries which represent violation witnesses. The value of `content` is a sequence of zero or more normal segments and exactly one final segment at the end. The structure of `content` and its nested items are summarized in Table 3.

Semantics. Each violation witness represents a set of some program executions violating the specified property. The witness is considered to be *valid* if the set is nonempty.

Let us consider a violation witness with $n \geq 1$ segments. An execution is represented by this witness, if the execution can be divided into n parts such that, for each $1 \leq i \leq n$, the i-th part matches the corresponding segment of the witness. An execution part matches a normal segment if

- it does not pass any avoid waypoint of the segment,
- it ends in the moment when the sequence point corresponding to the follow waypoint of the segment is entered for the first time in the execution part, and
- the follow waypoint is passed in this moment.

The final execution part matches the final segment if

- it does not pass any avoid waypoint of the segment and
- it violates the considered property during execution of the statement identified by the target waypoint.

Moreover, the execution must not contain any instruction that causes undefined behavior. An exception to this are witnesses of undefined behavior, in

Table 3: Structure of the `content` part of entries representing violation witnesses

Key	Value	Description
`content`	sequence	a sequence of zero or more normal `segment` elements and one final `segment` at the end
		description of `segment`
`segment`	sequence	a sequence of zero or more `waypoint` elements with `action: avoid` and one `waypoint` with `action: follow` at the end; the final segment ends by `waypoint` with `type: target`
		description of `waypoint`
`waypoint`	mapping	a basic building block of violation witnesses
		content of `waypoint`
`action`	`follow`	the waypoint should be passed through
	`avoid`	the waypoint should be avoided
`type`	`assumption`	restriction on variable values given by an expression
	`branching`	restriction specifying the result of a branching
	`function_enter`	restriction saying that a function is entered
	`function_return`	restriction on the result of a function call
	`target`	identification of a location of the property violation
`location`	mapping	the location of the waypoint; see Table 2
`constraint`	mapping	the constraint of the waypoint; not allowed with `type: function_enter` and `type: target`
		content of `constraint`
`format`	`c_expression`	for `type: assumption`, constraints are C expressions
	`acsl_expressions`	for `type: function_return`, constraints are specific ACSL expressions; not allowed for other `type` values
`value`	scalar	the actual constraint

which case the only instruction that causes undefined behavior must be the one represented by the target waypoint.

In each execution part, only the waypoints of the corresponding segment are evaluated. An `assumption` waypoint is evaluated at the sequence point immediately before the waypoint location. The evaluation must not lead to undefined behavior; otherwise the witness is incorrect. The waypoint is passed if the given constraint evaluates to true. A `branching` waypoint is evaluated at the sequence point immediately after evaluation of the controlling expression of the corresponding branching statement. The waypoint is passed if the resulting value of the controlling expression corresponds to the given constraint. A `function_enter` waypoint is evaluated at the sequence point immediately after evaluation of all arguments of the function call. The waypoint is passed without any additional constraint. A `function_return` waypoint is evaluated immediately after evalua-

tion of the corresponding function call. The waypoint is passed if the returned value satisfies the given constraint.

3 Evaluation

This section presents experiments with validation of verification results using both formats (1.0 and 2.0) to answer the following research questions:

- **RQ 1:** How does the performance of the validation of the new witness format compare to the old witness format?
- **RQ 2:** Does the new format improve attributes related to readability when compared to the old format?

In the experiments, the following tools were used.

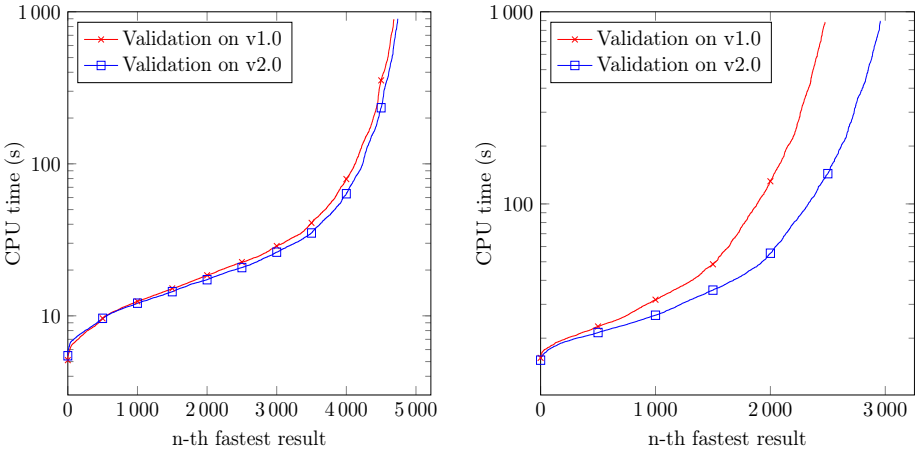
- CPAchecker [13, 27] is a verifier and witness validator that can produce and validate both correctness and violation witnesses in both formats. The experiments are based on version `0af0e41240`.
- Symbiotic [28] is a verifier that can produce violation witnesses in both formats. We use version `9c278f9`.
- Symbiotic-Witch2 [15] is a witness validator for violation witnesses in the old format (1.0). The experiments are based on version `svcomp24`.
- Witch3 [29] is a new witness validator based on similar principles as Symbiotic-Witch2, but designed for violation witnesses in format 2.0. The tool is made of Symbiotic in version `b011ec9` and Witch-Klee in version `6dabb94`.
- UAutomizer [30] is a verifier and witness validator that can produce both correctness and violation witnesses in both formats and validate correctness witnesses in both formats and violation witnesses only in format 1.0. We use version `0.2.4-?-8430d5a-m` and version `0.2.4-dev-0e0057c` for validation of YAML and GraphML correctness witnesses respectively.

Note that the support of the new witness format in all mentioned tools except UAutomizer has been implemented by authors of this paper.

For the experiments, we used all verification tasks of SV-COMP 2024 where the property to be verified is *unreachabilty of error function*, i.e., the specification used in Figs. 1 and 2. We did not use the witnesses produced during the competition [31], but rather based our experiments on fresh witnesses produced by the latest versions of Symbiotic and CPAchecker; the results are much better compared to the results from SV-COMP 2024 [32].

Benchmark Environment. For conducting our evaluation, we use BenchExec to ensure reliable benchmarking [33]. All benchmarks are performed on machines with an Intel Xeon E5-1230 CPU (4 physical cores with 2 processing units each), 33 GB of RAM, and running Ubuntu 22.04 as operating system. Each verification and witness validation task is executed with resource limits used in SV-COMP, i.e., 900 s of CPU time[3], 15 GB of memory, and 1 physical core (2 processing units).

[3] Except violation witness validation, where the convention is to use 90 s of CPU time.



(a) Validation using CPAchecker **(b)** Validation using UAutomizer

Fig. 3: Correctness witnesses produced by CPAchecker: Quantile plots for the time taken for validation of the old and new witnesses for two different validators

3.1 Evaluation Results for RQ 1 (Validation Performance)

One of the most important questions is whether the validation using the new format is as effective and efficient as with the old witness format.

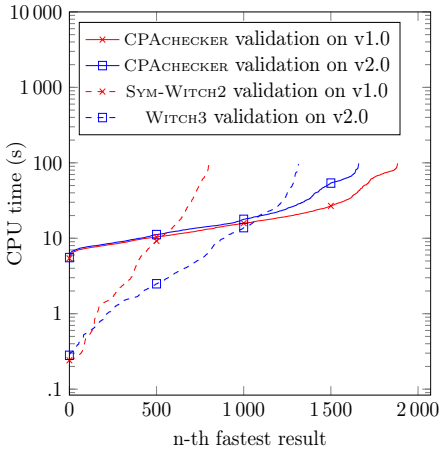
Correctness Witnesses. CPAchecker can generate correctness witnesses in both formats. The witnesses from CPAchecker were then validated by CPAchecker and UAutomizer. This allows for a direct comparison as shown in Fig. 3. We can observe in Fig. 3a that the validation performance of CPAchecker is largely identical when comparing both formats. This is to be expected, as the only thing that CPAchecker extracts from the GraphML witnesses are the invariants and their locations, and this is also the information that is present in and extracted from the witnesses in format 2.0.

For UAutomizer, the new format 2.0 substantially improves both the speed of validation and the number of witnesses that can be validated. Besides aiding in verification of the original property during validation, a witness can also add additional obligations for the validator to validate. This is the case here, where the extensive automaton that is embedded into CPAchecker's witnesses in format 1.0 is harder to prove correct for UAutomizer than the much simpler set of invariants that is present in the witnesses in format 2.0. Table 4 shows numbers of confirmed and refuted witnesses.

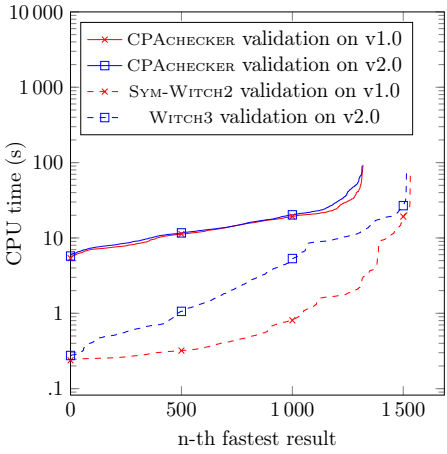
Violation Witnesses. We present the results of our evaluation regarding RQ 1 for violation witnesses in Fig. 4, which is complemented by Table 5 with the concrete number of validated and refuted witnesses. For witnesses generated by CPAchecker (cf. Fig. 4a), Witch3 is able to confirm significantly more witnesses in the new format than Symbiotic-Witch2 is able to confirm in the old

Table 4: Correctness witnesses produced by CPAchecker: Validation with CPAchecker and UAutomizer

		Witnesses v1.0		Witnesses v2.0	
Validator	Witnesses	Confirmed	Refuted	Confirmed	Refuted
CPAchecker	6 729	4 685	0	4 741	0
UAutomizer	6 729	2 478	109	2 959	2



(a) Validation of violation witnesses generated by CPAchecker



(b) Validation of violation witnesses generated by Symbiotic

Fig. 4: Violation witnesses: Quantile plots for the time taken for validation of the old and the new witnesses generated by two different verifiers for two different validators

format. Due to the large number of features and underspecified semantics of the GraphML format, Symbiotic-Witch2 does not support all the attributes used in the GraphML witnesses. Ignoring these features leads to a larger state-space that needs to be explored during validation, which results in more timeouts, or misinterpreting information in the witness and missing the described error. This is not the case for Witch3 as it supports the full set of features of the new format and makes use of all the information provided in the witness. For CPAchecker there is still a relatively small performance gap between validation with the new and old format. This is not surprising, as the GraphML-based format is inspired by the specification-automata language that CPAchecker uses internally, so achieving a similar performance requires still some more engineering.

For witnesses generated by Symbiotic (cf. Fig. 4b), we can observe that the number of new witnesses confirmed by Witch3 is almost the same as the number of old witnesses confirmed by Symbiotic-Witch2. Thus, both validation approaches are very close when it comes to effectiveness. This is also the case for

Table 5: Results of validating CPAchecker's and Symbiotic's **violation** witnesses with different validators; Witch stands for Symbiotic-Witch2 when validating old witnesses, and for Witch3 when validating new witnesses.

			Witnesses v1.0		Witnesses v2.0	
Verifier	Validator	Produced	Confirmed	Refuted	Confirmed	Refuted
CPAchecker	CPAchecker	2 011	1 880	35	1 657	9
CPAchecker	Witch	2 011	798	0	1 312	17
Symbiotic	Witch	1 556	1 533	5	1 516	0
Symbiotic	CPAchecker	1 556	1 319	29	1 315	27

Table 6: Different attributes of correctness witnesses in version 1.0 and 2.0 generated by CPAchecker

	Witnesses v1.0			Witnesses v2.0		
Attribute	Min	Median	Max	Min	Median	Max
Length in Lines of Code	53	1 536	1 014 533	18	28	1 058
Size in kB	3	52	35 573	1	1	965
Number of Nodes	3	114	26 899	-	-	-
Number of Edges	2	198	142 016	-	-	-
Number of Invariants	0	1	162	0	1	104

CPAchecker, which manages to confirm almost the same number of witnesses in both formats.

The new format for correctness witnesses does not reduce validation performance, for UAutomizer it shows a significant advantage over the old format. For violation witnesses, Witch3 handling the new format performs better than Symbiotic-Witch2 on the old format, while there is still room for improvement of CPAchecker as it performs slightly better on the old format.

3.2 Evaluation results for RQ 2 (Witness Readability)

Another important question is concerned with the attributes corresponding to the readability of the files encoding the witnesses. In particular, we are interested in the size and length of the witnesses, since this has a large effect on how easy they are to be read and understood by humans and machines.

Table 6 provides an overview of different attributes of the two versions of witnesses produced by CPAchecker for correctness. Table 7 does the same for violation witnesses produced by CPAchecker and Symbiotic. Some attributes are only applicable to one of the two versions of witnesses.

For correctness witnesses we can see that the new witnesses are usually very small in comparison to the old witnesses. This is because the new witnesses encode only the invariants, while the old witnesses encode information about the control-flow of the program. One explanation for the difference is that witnesses in version 1.0 roughly scale with the size of the program. While witnesses in version 2.0 scale

Table 7: Different attributes of violation witnesses in version 1.0 and 2.0 generated by CPAchecker and Symbiotic

	Witnesses v1.0			Witnesses v2.0		
Attribute	Min	Median	Max	Min	Median	Max
Length in Lines of Code	12	372	258 730	27	171	114 460
Size in kB	2	14	9 098	1	6	3 071
Number of Nodes	1	38	28 304	-	-	-
Number of Edges	0	42	28 793	-	-	-
Number of Waypoints	-	-	-	1	13	9 537

only with respect to the amount of invariants, which for CPAchecker is roughly correlated to the amount of function calls and loops. As we saw in Sect. 3.1, this extra information is not necessarily relevant for validation.

For violation witnesses, we see that, apart from a small factor due to overhead in describing the automaton, both formats are similar in all metrics. This is not surprising, as both formats encode similar information about an error path. Therefore, they both roughly scale with the amount of assumption for nondeterministic variables and the amount of branching decisions in the error path.

The tables show that the new witnesses are usually much shorter than the old witnesses. As we have seen in Sect. 3.1, this does not have a negative impact on the validation performance, since the information most relevant for validation is retained. Having less information makes it much easier for a verification engineer to understand the witness and use it in some further processing steps.

In summary, witnesses in version 2.0 are generally much smaller and easier to read than witnesses in version 1.0, while retaining all important data.

3.3 Threats to Validity

Internal Validity. We used the benchmarking framework BenchExec [33] to run the experiments, which uses the most modern Linux features for reliable benchmarking. This tool also makes sure to never run two different executions on the same physical core, in order to avoid interference of shared computing resources. Our validation tools might contain bugs, which could lead to wrong conclusions, however, our claim is that the new format works already sufficiently well to serve as an alternative format.

External Validity. The conclusions about the validators might not hold for other validators that will be developed in the future, also, witnesses generated by other verifiers might have different characteristics. However, other tools are not expected to deviate much from the presented witnesses, because they would serve the same purpose of testifying the bug or proof. Our experiments were done on a

large benchmark set, which is also used in competitions, but it could still be the case that there are witnesses and programs for which the results presented are not applicable. Since extending and improving the witness format is an ongoing process, we expect that if this is the case, it will be adequately addressed in the future.

4 Version 1.0 vs. Version 2.0

The witness format 2.0 is closely tied to the actual program syntax. While the format 1.0 uses an automaton largely independent of the program syntax and closely tied to the program representation as control-flow automata internally used by some verifiers. Due to this, the format 2.0 is more succinct, has well-defined semantics, and is easier to understand by humans. On the other hand, format 1.0 is more expressive, for example it can define different loop invariants for the same loop, when two different paths are taken to reach the loop.

Currently, the format 2.0 has the same practical limitations as the format 1.0. In the case of correctness witnesses, they have not yet been defined for concurrency safety, memory safety and for termination. Violation witnesses have not yet been defined for concurrency safety. There are also features which are not yet supported by the new format but which are straightforward extensions, such as support for Java and violation termination witnesses. Extending witnesses to be able to validate more programs and specifications is ongoing work, we expect that the simplification of the syntax and clarification of the semantics with format version 2.0 will make it easier to extend the format in the future.

In order to validate our concept of the new format, we reported our initiative to the SV-COMP community, and the jury made a decision to immediately include the new format as an alternative to the existing format, in order to quickly adopt it and improve the state of the art. This was seen in SV-COMP 2024 [32], where 8 verifiers and 4 validators supported the correctness witnesses v2.0 and 2 verifiers and 2 validators supported the violation witnesses v2.0. Table 8 shows all these tools and their support of witnesses formats in detail.

This also shows the large interest the software verification community has in the new format, since the first mention of the format for correctness witnesses was only in September 2021[4] and the work on the violation witnesses part of the new format started only in April 2023.

5 Conclusion

Verification witnesses are an important part of the software-verification ecosystem. Just like verification tools, specification formats, and witness validators, there is also a need to improve the *format* for verification witnesses. This paper introduces the witness format version 2.0, which changes the container format from GraphML to YAML, has more concise data representation, and has a clearly

[4] https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/merge_requests/44

Table 8: Tools with some support of witnesses format 2.0 and their abilities to generate/verify correctness/violation witnesses in format 1.0/2.0 in SV-COMP 2024; tools where the support of witness format 2.0 was implemented by the authors of this paper are typeset in bold

	Witness Generation				Witness Validation			
	Correctness		Violation		Correctness		Violation	
Tool	v1.0	v2.0	v1.0	v2.0	v1.0	v2.0	v1.0	v2.0
CPAchecker	•	•	•	•	•	•	•	•
Symbiotic	•		•	•				
Symbiotic-Witch2							•	
Witch3								•
UAutomizer	•	•	•		•	•	•	
UKojak	•	•	•					
UTaipan	•	•	•					
UGemCutter	•	•	•					
Mopsa	•	•				•		
CPV	•	•	•					
Goblint	•	•				•		

defined semantics independent from control-flow automata. Besides describing the syntax and semantics of the new format, we also evaluated the effectiveness and efficiency induced by the new format. In sum, the new witnesses are much smaller and the experimental results show a significantly improved confirmation rate for some validators: using the new format, UAutomizer can confirm 481 more correctness witnesses (Table 4) and Witch3 can confirm 514 more violation witnesses (Table 5). Furthermore, shortly after we proposed this new format, already seven other tools support the format, which is an indicator that the developers value the new format.

Data-Availability Statement. A reproduction package (that includes all software and data that we used for our experiments) is available on Zenodo [34].

Funding Statement. P. Ayaziová and J. Strejček were supported by the Czech Science Foundation grant GA23-06506S. D. Beyer, M. Lingsch-Rosenfeld, and M. Spiessl were supported by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 496588242 (IdeFix).

References

1. 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
2. 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

3. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
4. Beyer, D., Strejček, J.: Case study on verification-witness validators: Where we are and where we go. In: Proc. SAS. pp. 160–174. LNCS 13790, Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_8
5. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
6. 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
7. Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: Proc. ICSE. pp. 536–548. ACM (2022). https://doi.org/10.1145/3510003.3510064
8. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Proc. CAV (2). pp. 502–509. LNCS 9780, Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_28
9. 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
10. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM. p. 111–128. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
11. 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
12. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing. pp. 501–512. LNCS 2265, Springer (2001). https://doi.org/10.1007/3-540-45848-4_59
13. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
14. Chalupa, M., Rechtáčková, 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
15. Ayaziová, P., Strejček, J.: Symbiotic-Witch 2: More efficient algorithm and witness refutation (competition contribution). In: Proc. TACAS (2). pp. 523–528. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
16. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009
17. Turing, A.: Checking a large routine. In: Report on a Conference on High Speed Automatic Calculating Machines. pp. 67–69. Cambridge Univ. Math. Lab. (1949), https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8
18. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_6
19. Sternagel, C., Thiemann, R.: The certification problem format. In: Proc. UITP. pp. 61–72. EPTCS 167, EPTCS (2014). https://doi.org/10.4204/EPTCS.167.8

20. Järvisalo, M., Berre, D.L., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine **33**(1) (2012)
21. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **1610**(06229) (October 2016)
22. Wetzler, N., Heule, M.J.H., Jr., W.A.H.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: Proc. SAT. pp. 422–429. LNCS 8561, Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31
23. Bury, G., Bobot, F.: Verifying models with Dolmen. In: Proc. SMT Workshop. CEUR Workshop Proceedings, CEUR (2023)
24. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Proc. ASE. pp. 200–205. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115633
25. OASIS: Static analysis results interchange format (sarif) version 2.0 (2019)
26. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
27. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
28. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
29. Ayaziová, P., Strejček, J.: Witch 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
30. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate automizer and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS. LNCS , Springer (2024)
31. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2024). Zenodo (2024). https://doi.org/10.5281/zenodo.10669737
32. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
33. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
34. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Reproduction package for SPIN 2024 article 'Software verification witnesses 2.0'. Zenodo (2024). https://doi.org/10.5281/zenodo.10826204