Non-termination Witnesses and Their Validation

Zsófia Ádám [1], Paulína Ayaziová [3], Levente Bajczi [1], Dirk Beyer [2]✉, Marek Jankola [2], Marian Lingsch-Rosenfeld [2], and Jan Strejček [3]

[1]Department of Artificial Intelligence and Systems Engineering, Budapest University of Technology and Economics, Budapest, Hungary
[2]Institute for Informatics, LMU Munich, Munich, Germany
[3]Faculty of Informatics, Masaryk University, Brno, Czech Republic

***Abstract*—Designing algorithms for complex problems as certifying algorithms is an important approach to ensure correctness of computational results. Instead of producing an output y for an input x, a certifying algorithm produces as output for x not only y but also a witness w. The witness w (also called certificate) can now be used to check that y is indeed the correct output for input x. Witnesses and their validation also exist in the area of automatic software verification, and a large number of tools support verification witnesses. SV-COMP 2025 reports 62 verifiers producing witnesses and 18 tools for witness validation. In 2023, a new version 2.0 of the witness format for software verification was introduced to overcome several problems with the previous format, and this new format is now widely supported. However, there is no format with a clear definition and semantics for witnesses of non-termination. This paper closes this gap by presenting an extension of the witness format 2.0 to support program non-termination. Besides explaining the design of this extension, we describe various approaches to generate and validate non-termination witnesses. We also give an overview of current tool support of the extended format, i.e., the verifiers that can generate non-termination witnesses and the witness validators able to analyze these witnesses. Finally, we present an experimental evaluation showing the performance of these tools on program-termination tasks of SV-COMP 2025.**

***Index Terms*—Verification Witness, Software Verification, Validation, Exchange Format, Non-termination, Counterexample**

I. Introduction

The task of automatic tools for software verification is to decide whether a given program satisfies a given property. Negative answers are traditionally accompanied by a *counterexample*, which is a description of a program execution that violates the property. The first generic exchange format for counterexamples, also called *violation witnesses*, was introduced in 2015 [1]. This standardized GraphML-based format allowed for the validation of violation witnesses and was soon adopted by the *Competition on Software Verification* (SV-COMP) [2]. The format was quickly extended to cover also *correctness witnesses*, i.e., the arguments for the decision that the program satisfies the property [3]. The final version of the format [4], now called witness format 1.0, supports many program features and properties, including program termination.

Several years ago, the community around SV-COMP identified some serious drawbacks of the format version 1.0. The main drawback is that its semantics is formulated over programs represented as *control-flow automata* (CFA). This makes the semantics over real programs ambiguous as there is no commonly accepted transformation of real programs to CFAs, and existing transformations differ in several aspects, such as granularity of instructions on edges or handling of function calls. Other identified issues are, for example, that some features of the format are unused in practice and not supported by any tools, or that there are only vague descriptions of some parts of the format, including witnesses of program non-termination.

In reaction to the drawbacks, a new witness format 2.0 [5] was introduced in 2023 and quickly adopted by the community [6]. This YAML-based format has a semantics defined directly on C programs, and its principles are easily adoptable to any imperative language. The initial version of format 2.0 supports only some basic safety properties: unreachability of a given error function, reachability of signed arithmetic overflow, and unreachability of an invalid pointer dereference. One of the most important properties currently not supported by the witness format 2.0 is termination.

In this paper, we focus on counterexamples to termination, i.e., *termination violation witnesses*, which can be also called *non-termination witnesses*. Note that a program execution can be non-terminating basically for two reasons:

1) it is blocked by some instruction (e.g., it waits endlessly for some event that will never happen) or
2) it runs forever (i.e., evaluates an infinite sequence of instructions, typically corresponding to the repeated evaluation of a program loop).

The programs with non-terminating executions of the first kind can be detected by a suitable reachability analysis. In this paper, we consider solely the non-terminating executions of the second kind.

We first recall background and related work relevant to (non-) termination program analysis (Sect. II). Then, we introduce the extension of witness format 2.0 to support non-termination witnesses and explain our design choices (Sect. III). We extended several state-of-the-art open-source verification tools to generate non-termination witnesses in the extended format, namely, Symbiotic [7], Theta [8], and TransVer [9]. We also extended several state-of-the-art open-source witness validation tools to analyze these witnesses, namely, CPAchecker [10] Theta, and Witch [11]. We describe the approaches of individual verification (Sect. IV) and validation (Sect. V) tools to generate and analyze the witnesses, respectively. Finally, we present an experimental evaluation of these tools on non-terminating programs of SV-COMP 2025 (Sect. VI).

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00164

Contributions. We make the following contributions:

- an extension to the witness format 2.0 to support non-termination witnesses; in contrast to the old witness format 1.0, the extension offers non-termination witnesses with clearly specified syntax and semantics,
- implementations of the export and validation of non-termination witnesses in the extended format in multiple state-of-the-art tools based on different approaches,
- an evaluation of these tools on all relevant benchmark tasks of SV-COMP 2025 [12], and
- a set of handcrafted witnesses [13], written in the extended witness format, in order to encourage and facilitate its adoption by the community.[1]

Further, we integrated our proposal with all other current proposals for extensions of the witness format 2.0 [14, 15], developed their joint documentation and contributed to the release of version 2.1 of the format and its linter.

II. Background and Related Work

A large body of research results and literature is available to support the foundation and inspiration of our work. We provide a short overview of the area below.

Verification Witnesses. Our contributions build on the concept of certifying algorithms [16], which was first applied to graph algorithms. The concept of witnesses was introduced to software verification in 2013 [17], and the first standard exchange format based on GraphML was proposed in 2015 [1]. The format was later extended to include correctness witnesses [3] and witnesses for concurrent programs [18]. Verification witnesses were also used for deriving test cases [19] and for debugging [20]. An overview is also provided in the literature [4]. The latest development was the new format version 2.0 [5], which is based on the YAML format. Extensions were proposed to include ghost variables to witness the correctness of concurrent programs [14] and to include function contracts [15]. The community around the termination competition (termCOMP [21]) has developed the CPF [22] format for certifying termination of term-rewriting systems.

Termination. Program termination is one of the most important properties to be mentioned in specifications for software, considered for a long time in computer science [23, 24, 25]. Proving termination was made practically relevant by the seminal papers by Podelski and Rybalchenko [26, 27] and made applicable to industry shortly afterwards [28, 29]. The problem of termination, if restricted to finite systems, can be reduced to a reachability problem [30].

Non-termination. It is also important to consider proving that a program does not terminate (i.e., it has some infinite run), and there is a large set of verification tools supporting this. There are 16 participants of SV-COMP 2025 with positive scores in the category for program termination[2] and 11 of them decided that some programs have non-terminating runs. We refer to the competition report [12] for obtaining references to literature about those tools. By using reductions from termination to reachability, even bounded model checkers (e.g., [31]) or test-generation tools such as fuzzing tools (e.g., [32]) can be used to prove non-termination.

The GraphML-based witness format [1] was extended for non-termination of C programs (https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/tree/svcomp18/termination), and this extension was soon adopted by several verification and validation tools and by SV-COMP since 2018 (https://sv-comp.sosy-lab.org/2018/rules.php). While the extension of the format for non-termination witnesses and the fact that it served as the community standard for many years is a great achievement, it suffers from some serious problems (partly inherited from the original GraphML-based witness format): some requirements on the structure of non-termination witnesses are not clearly stated, and the semantics over real programs is ambiguous. As a consequence, some tools produce or even validate non-termination witnesses that do not follow the format.[3]

Our work extends the witness format 2.0 to accommodate non-termination witnesses. We aim for a clear description of the extension, unambiguous semantics, and providing useful information, even in minimal non-termination witnesses.

Approaches to Proving Non-termination. The approaches that researchers have tried out so far include *recurrent set* construction [31, 35, 36, 37], *loop acceleration* [38, 39], and *program-reversal* techniques [40]. Some of our tool extensions use an approach based on lassos with recurrent sets.

Lassos. One of the standard approaches [41, 42, 43] to prove non-termination for a given program is to find a *lasso* with a *recurrent set* [35] of program states. A lasso for a given program p consists of two parts. The first part, called *stem*, is a finite path of p that starts at the initial program location and leads to a program location l. The second part, called *loop*, is a finite path of p (with at least one transition) leading from l back to l. The lasso is *feasible* if there exists an execution along the stem followed by infinitely many copies of the loop. To prove that a given loop is feasible, it is sufficient (but not necessary) to find a *recurrent set* R of program states at program location l such that at least one state of R is reachable from the initial program location by executing the stem and from each program state of R it is possible to reach a program state from R again by executing the loop. The recurrent set R is often represented by a formula called *recurrence condition*.

III. Non-termination Witness Format

Before we introduce the format extension, we briefly recall the shape of violation witnesses in format 2.0. The details including the precise syntax and full semantics can be found in the corresponding paper [5]. We then describe the format extension along with our design decisions and show the improved informative quality of even minimal witnesses.

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1634

[2] https://sv-comp.sosy-lab.org/2025/results/results-verified/META_Termination.table.html and [12, Tables 10–11]

[3] For example, the format [4] explicitly requires that each termination violation witness must contain at least one node marked as `cyclehead`, but AProVE [33] and Bubaak [34] produce non-termination witnesses without any such node and some of them are even validated by some witness validators.

Note that the format 2.0 can describe only violation witnesses of specific safety properties. A program violates a safety property if it has an execution that enters an unsafe program state. A violation witness in format 2.0 describes a set of some program executions violating the considered property and the witness is considered to be *valid* if the set is non-empty.

Structure of Violation Witnesses 2.0. A *violation witness* in format 2.0 is a finite sequence of *segments*, where each segment consists of a sequence of finitely many *waypoints*. Each waypoint has a `type`, an `action`, and it is associated with a program `location`. Some types of waypoints also have a `constraint`. There are five possible waypoint types:

- an `assumption` waypoint claims that a given `constraint` is satisfied immediately before the statement or declaration identified by the `location`,
- a `branching` waypoint must be associated with a branching statement and its `constraint` claims that the branching condition is evaluated to either `true` or `false`,
- a `function_enter` waypoint is associated with a function call and claims that the called function is entered. Waypoints of this type have no `constraint`,
- a `function_return` waypoint is also associated with a function call and claims that the function call has been evaluated and that the returned value satisfies the given `constraint`, and
- a `target` waypoint claims that the associated statement or full expression contained in the statement violates the considered safety property. Waypoints of this type have no `constraint`.

A waypoint's `action` is one of two possible values: (1) `follow` indicates that the waypoint should be passed, while (2) `avoid` indicates that the waypoint should be avoided. Waypoints with the action `avoid` cannot be of type `target`.

Each segment is a finite sequence of waypoints with action `avoid` terminated by one waypoint with action `follow`. A segment with a `follow` waypoint of type `target` is called *final* and all other segments are called *normal*. A violation witness is a finite sequence of normal segments terminated by one final segment.

Semantics of Violation Witnesses 2.0. An execution is represented by a witness with n segments if it can be divided into n parts, where for every $1 \leq i \leq n$, the i-th part matches the i-th segment. An execution part matches a *normal* segment if

- it does not pass any waypoint with action `avoid` of the segment,
- it ends in the moment when the `follow` waypoint of the segment is passed, and
- the evaluation point of this `follow` waypoint is not visited before by this execution part.

An execution part matches the *final* segment if:

- it does not pass any waypoint with action `avoid` of the segment, and
- it enters an unsafe program state during evaluation of the expression or statement pointed by the `target` waypoint.

The definition of passing a waypoint corresponds to the intuitive meaning given above. We say that an execution passes a waypoint if it visits its corresponding evaluation point and the requirements given by its constraint are satisfied. For example, an `assumption` waypoint is passed by an execution that enters the associated `location` and the current program state satisfies the `constraint` of the waypoint.

Note that `avoid` waypoints of a segment are evaluated (but must not be passed) each time their evaluation point is reached during the execution part, regardless of their order in the segment, i.e., they can be evaluated in any order an arbitrary number of times.

Design Decisions. The witness format 2.0 can only describe finite traces and representing infinite executions requires an extension of this format.

We had to decide whether our non-termination witnesses will contain recurrence sets. At the end, we rejected this idea for several reasons. First, a recurrent set can put a specific constraint on the content of dynamically allocated memory (e.g., that a dynamically allocated list is cyclic or descending) and on the call-stack content, while currently there are no established formalisms to efficiently handle such constraints in witnesses. Second, verification techniques able to identify a non-terminating program execution are not necessarily able to easily derive the corresponding recurrence condition. Finally, validating the fact that some condition is a recurrence condition can be expensive as one has to show that from *every* state satisfying the condition there exists an execution path to a state satisfying the same condition, whereas for validation of program non-termination this might only be necessary for *reachable* states.

We were looking for a minimal extension of the witness format 2.0 (in the terms of both syntax and semantics) that would allow us to represent the (abstracted) stem and loop of non-terminating program executions.

Structure and Semantics of Non-termination Witnesses. We propose to extend the format with a new waypoint action `cycle` which indicates that the waypoint has to be passed infinitely often. Non-termination witnesses then use the above defined *normal* segments and *cycle* segments, which are finite sequences of `avoid` waypoints terminated by one waypoint with action `cycle`. A non-termination witness is then a finite sequence $n_1 n_2 ... n_i c_1 c_2 ... c_j$ of $i \geq 0$ normal segments n_1,...,n_i followed by $j > 0$ cycle segments c_1,...,c_j.

Such a witness represents every infinite program execution that can be divided into infinitely many non-empty parts such that, for each k, the k-th execution part matches the k-th segment of the sequence $n_1 n_2 \dots n_i (c_1 c_2 \dots c_j)^\omega$, where the definition of an execution part matching a *cycle* segment is the same as the definition of matching a *normal* segment given above. The witness is valid if it represents at least one execution of the program for which it was produced.

The sequences $n_1 n_2 ... n_i$ and $c_1 c_2 ... c_j$ roughly correspond to the stem and the loop of infinite executions. However, note that each of the two sequences can represent many different execution parts and thus the executions represented by such a witness do not have to be ultimately periodic. In fact, they do not even have to correspond to a program loop (it can also represent, for example, an unbounded recursion).

```c
int main() {
  int i = nondet();
  while (i > 0) {
      if (i != 5) {
          i = i-1;
      }
  }
  return 0;
}
```

```yaml
- entry_type: "violation_sequence"
  metadata: ...
  content:
  - segment:
    - waypoint:
        type: "assumption"
        constraint:
          value: "i == 5"
          format: "c_expression"
        location:
          file_name: Ex02_sim.c
          line: 3
        action: "follow"
  - segment:
    - waypoint:
        type: "branching"
        constraint:
          value: "true"
        location:
          file_name: Ex02_sim.c
          line: 3
        action: "cycle"
```

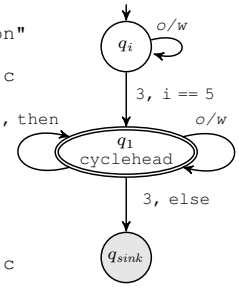

Fig. 1: Program Ex02_sim.c (top, based on Ex02.c from SV-Benchmarks) with a non-termination witness in format 2.1 (bottom left, based on Ex02.good1...yml from SV-Benchmarks) and an equivalent witness in format 1.0 (bottom right)

```yaml
- entry_type: "violation_sequence"
  metadata: ...
  content:
  - segment:
    - waypoint:
        type: "assumption"
        constraint:
          value: "1"
          format: "c_expression"
        location:
          file_name: ...
          line: 123
        action: "cycle"
```

Fig. 2: Trivial non-termination witness in format 2.1 (left) and in format 1.0 (right); the witness in format 2.1 contains more information about the non-terminating behavior

Examples. An example of a simple non-termination witness is provided in Fig. 1. The witness consists of one normal segment and one cycle segment, both without any avoid waypoints. While both waypoints refer to the same location, the `assumption` waypoint is evaluated before the execution reaches the `while` loop at line 3, and the `branching` waypoint is evaluated after every evaluation of the controlling expression of the loop. The witness then represents exactly one execution of the given program, where the value of `i` before the evaluation of the `while` loop is `5` and the while loop never terminates. We also provide an equivalent witness in format 1.0, which is a Büchi automaton with edges matching the program statements. A precise description of the syntax and semantics of non-termination witnesses in format 1.0 is, up to our best knowledge, not available.

To encourage adoption by the community, we contribute a small set of handcrafted example witnesses that can also be used as a test suite for developing a witness validator. The example set contains two programs and 17 syntactically correct witnesses (8 valid and 9 invalid), including the example described above. All examples are contributed to the SV-Benchmarks repository and published on Zenodo [13].

Minimal Non-termination Witnesses. In order to fulfill a requirement to produce a verification witness, some verification tools output simple witnesses that provide almost no information. Figure 2 shows a minimal non-termination witness in format 1.0 and format 2.1. While the minimal witness in format 1.0 does not provide any useful information, except the claim that the program is cycling, each non-termination witness in format 2.1 has to contain at least one cycle segment and the segment has to contain one waypoint with action `cycle`. In Fig. 2, this waypoint is of type `assumption` with a tautological `constraint`. Still, the waypoint has to determine the corresponding location (in the given witness, it is identified by the program line `123` in the specified file). Each program execution represented by this witness has to enter this location infinitely often. This means that even a minimal witness in the new format has to provide some relevant information about program non-termination.

IV. Construction of Non-termination Witnesses

We have implemented the support for the proposed extension to the witness format 2.0 in multiple verifiers. In the following sections, we first describe the common concepts that occur in most of the approaches and then the individual techniques of these tools with focus on the proposed witness format.

Liveness-to-Safety Reduction. Most of the verification and validation approaches implemented in the presented tools follow the same concept of reducing liveness checking to safety checking [30]. In our setting, we reduce the verification of non-termination, which is a liveness property, to the verification of a safety property: reachability. Conceptually, we add additional logic to the program or to the verification algorithm to record a visited concrete state at the loop head. The corresponding safety property then checks that each subsequently visited state is different from the saved one.

In practice, tools typically introduce an additional ghost variable v' for each variable v modified inside the loop. The values of v can be non-deterministically saved into v' upon each visit to the loop head. The property $\bigvee_{v \in LVar} v \neq v'$, where $LVar$ is the set of variables modified in the loop, must then hold after each loop iteration. If this property is violated, it indicates the existence of a non-terminating execution that revisits the same concrete state, saved in v', infinitely often.

This reduction can be implemented either at the program level or encoded directly into SMT queries posed by the verification algorithm. The approach is sound but incomplete for programs with infinite state spaces.

Symbiotic. Symbiotic [7] is a program-analysis tool that combines configurable instrumentation, program slicing,

and symbolic execution. Its termination analysis uses the liveness-to-safety reduction, slicing, and symbolic execution to verify assertion safety of the transformed program [44].

The analysis begins by instrumenting trivial infinite loops, such as `while(1){}`, with failing assertions. The program is then reduced by slicing [45] that uses *non-termination sensitive control dependence* [46] and considers exit points of the program and the added assertions as slicing criteria. SYMBIOTIC then processes non-trivial loops that modify only program variables and have a single entry. It applies a simplified liveness-to-safety reduction by instrumenting the loop head such that upon every visit, the current state is stored in the ghost variables, and instrumenting the loop end with the corresponding assertion. Hence, the assertion is violated only if the same state repeats after one (but not necessarily the first) iteration of the loop.

This instrumented program is then explored by symbolic execution to find possible violations of the instrumented assertions. If such an error is found, SYMBIOTIC reports the program to be non-terminating.

We extend SYMBIOTIC to produce witnesses in the proposed format, utilizing its ability to generate witnesses for assertion violations. Upon finding an assertion violation, SYMBIOTIC obtains a vector of inputs that led the execution to the failed assertion. As this vector does not contain the return values of input functions that were removed by slicing, the process is repeated without slicing, this time using the input vector to significantly reduce the state space of the program. After rediscovering the violation, a new input vector is generated and converted into a witness based on `function_return` waypoints specifying the inputs.

In our extension, if the failed assertion corresponds to a trivial infinite loop, all inputs are turned into waypoints in normal segments and the only cycle segment contains an `assumption` waypoint with `constraint` `"1"` and a location pointing into the loop. If the failed assertion corresponds to a non-trivial loop, then the inputs read before the last iteration of this loop are turned into waypoints in normal segments and the inputs read during the last iteration before the assertion violation are turned into waypoints in cycle segments. If there are no inputs of the latter kind, we use the same cycle segment as in the case of trivial infinite loops. The produced witnesses do not contain any `avoid`-action waypoints.

THETA. The verification framework THETA [47] is primarily built around symbolic model checking. Its modular architecture enables a diverse set of verification algorithms to work on a wide selection of input formats, including C programs [48, 49]. It supports three different verification approaches for determining termination:

1) A lasso checker [50] using *counterexample-guided abstraction refinement* (CEGAR) with bounded unwinding [51];
2) A liveness-to-safety reduction for bounded techniques;
3) A transformation to a system of *constrained Horn clauses* (CHCs), then solving with a dedicated CHC solver [52].

These approaches also represent the three configurations of THETA in SV-COMP 2025 [12]: THETA for CEGAR-based analyses, EMERGENTHETA for bounded techniques, and THORN for delegating to CHC solvers.

1) CEGAR-Based Lasso Checking: The CEGAR loop in THETA supports both straight and lasso-shaped counterexamples [50]. Although the latter is primarily intended to verify LTL properties, we can easily use it for termination checking. To this end, we create a Büchi automaton accepting all infinite sequences, and the verification starts by abstracting the program with an initial coarse precision. It then builds the synchronous product of the abstract model and the specification automaton to search for accepting lassos. If no such shape exists, the program is terminating. If a potential counterexample is found, its feasibility is checked via a concretization step. Spurious counterexamples lead to refinement and iteration.

THETA supports Boolean and Cartesian predicate abstraction, as well as explicit-value abstraction [47]. We used explicit-value abstraction for the experiments in Section VI.

2) Bounded Techniques: The EMERGENTHETA configuration of THETA supports bounded model checking (BMC) [53], k-induction [54], interpolation-based model checking (IMC) [55], property directed reachability (PDR/IC3) [56], and multi-valued decision diagram (MDD) analyses [57]. These all work on symbolic transition systems, which can undergo a liveness-to-safety reduction to enable error state reachability algorithms to prove non-termination. The detected error trace is then transformed back into a lasso-shaped trace before writing it into a witness file. We primarily used k-induction for the experiments in Section VI.

3) Delegating to CHC Solvers: Constrained Horn clauses are proven to be a convenient yet powerful representation of a program-verification problem [58, 59]. Encoding a program and its safety property as constraints over predicates representing program locations is both straightforward to do, and a solution directly represents a useful abstraction of the system. Termination, without a liveness-to-safety reduction, can be encoded via an auxiliary *index* parameter in all predicates. In this parameter, we keep track of the number of edges traversed from the initial state (i.e., the length of the path from the initial state to the current state), and if we encounter the same state twice with different indices, the path leading through this state back to itself is a counterexample to the termination property.

The THORN configuration relies on this transformation, then solves the resulting system of CHCs using external CHC solvers such as ELDARICA [52].

THETA was already capable of exporting reachability witnesses, which we extended to export non-termination witnesses. Extending the CHC-solver-based method of THORN to termination was also added in this work.

Each configuration of THETA returns a sequence of actions and states as a counterexample, which correspond to statements in the program and the value of variables at certain sequence points of the program execution. While we use large-block encoding [60] on the edges of our CFA, we run a separate concretization pass on a large-block-encoded trace that splits it into smaller segments, each corresponding to a single C statement, thus also including a state with full variable information at most sequence points of the program.

To transform this counterexample trace into a non-termination witness in the proposed format, we first identify the first time the last state appears in the trace, which is designated

```
- entry_type: "violation_sequence"
  metadata: ...
  content:
  ...
  - segment:
    - waypoint:
        type: "branching"
        constraint:
          value: "true"
        location:
          file_name: ...
          line: 6
        action: "follow"
  ...
  - segment:
    - waypoint:
        type: "branching"
        constraint:
          value: "false"
        location:
          file_name: ...
          line: 6
        action: "follow"
  ...
```

```
1  int main() {
2    int i = nondet();
3    int i_1;
4    int saved = 0;
5    while (i > 0) {
6      if (nondet() &&
7          !saved) {
8        i_1 = i;
9        saved = 1;
10     } else {
11       assert(!saved ||
12              i != i_1);
13     }
14     if (i != 5) {
15       i = i-1;
16     }
17   }
18   return 0;
19 }
```

Fig. 3: Transformed program from Fig. 1 (right) with a snippet of an assertion-violation witness (left)

as the cycle head, marking the beginning of the lasso loop. We designate all states before the loop head to be part of the stem, thus becoming `follow`-action `assumption` waypoints, and all the states after (including the cycle head) to be part of the loop, thus receiving `cycle`-action `assumption` waypoints.

Currently, THETA does not use `avoid`-action waypoints, and only uses assumptions on the `follow`- and `cycle`-action waypoints. It is capable of determining branching and function-return information as well, but does not use these by default.

As THETA uses an inlined representation of the program, variable-scoping information needs to be re-established when THETA creates the assumptions for the witness. It needs to not include variables that have not been initialized at that point (even though it handles them as if their values were nondeterministically chosen), and it also needs to not include variables that do not appear in the corresponding scope of the C program. To overcome this, THETA only outputs variables with changed values since the last segment, which helps mitigate the latter problem, and determines which variables have been written —thus which to include— to solve the former problem.

TRANSVER. The program-transformation tool TRANSVER [9] implements a modular framework focused on reducing various specifications to reachability. It takes a C program as input and produces a transformed C program as output.

One of the supported transformations reduces non-termination to reachability according to the liveness-to-safety reduction. The transformed version of the program from Fig. 1 is shown in Fig. 3. TRANSVER does not yet support programs with arrays or dynamic memory allocation because storing a state with a complicated data structure can be very expensive for verification tools.

The reduction enables any reachability verifier participating in SV-COMP [12] to be applied to verify termination. However, the witnesses produced by the transformation-verifier chain are violation witnesses for a safety property, certifying the transformed program. An example of such a witness is shown in Fig. 3. In this work, we complete the workflow of this chain by transforming the produced witness to a non-termination witness in our extension of the format 2.0. To obtain a non-termination witness, we make the following major adjustments:

1) Identify the Cycle: The violation witness for reachability describes a finite execution that stores a state to the ghost variables at some point and then violates the assertion in the final state. We must identify the waypoint in the witness that indicates when the state is stored, and the waypoint that violates the assertion. By merging the two segments containing these waypoints, we enclose the infinite execution loop described by the witness.

The state that violates the assertion is described by the `target` waypoint in the original violation witness. There is exactly one such waypoint in witness version 2.0 [5]. The state that stores variable values into ghost variables can be identified as the `branching` waypoint with costraint `"true"` at the `if` condition guarding the violated assertion. If there is no such `branching` waypoint, TRANSVER will not produce a witness. For example, the witness in Fig. 3 contains two waypoints for the `if` condition on the line 6. The first has the constraint `"true"`, i.e., storing the state, and the second is followed by the `target` waypoint, hence violating the assertion and reaching the same state again. We transform the `follow` action of all waypoints between these two into `cycle`, and we remove all waypoints that follow the second `branching` waypoint.

2) Adjust the Locations: Some of the line numbers in the `location` entries are shifted due to the instrumented code, and some refer to the code that does not exist in the original program. We remove the `branching` waypoints that point to locations containing instrumented `if` conditions. We remove all the `avoid` waypoints. We retain the `assumption` waypoints that point to the instrumented code but do not refer to the instrumented variables, and we update them to point to the loop from the original program instead. We identify the original loop locations in the input program for the `cycle` waypoints and adjust them accordingly. Concerning the other waypoints, if we can determine their original source positions, which may not always be feasible due to the formatting changes during transformation, we also adjust and retain them.

V. VALIDATION OF NON-TERMINATION WITNESSES

We have also extended several witness validators to process non-termination witnesses in the suggested format. In this section, we describe the individual specifics of each validation approach.

CPACHECKER. The verifier CPACHECKER [10] and also its validator [61] gain its strength from composing different *Configurable Program Analyses* (CPA) [62] into a single algorithm. Each CPA tracks a different abstract domain that collects information about the concrete states that it represents. In our validation approach, we use three main CPAs: the automaton CPA [4], the termination CPA, and the predicate CPA [63].

We implemented the parsing of non-termination witnesses and the construction of an automaton. Each segment defines a state mapped to a program location. Transitions then connect the states in the sequence ordered according to the order of

the segments in the witness. There is an additional transition from the last state to the state corresponding to the first cycle segment, creating the lasso. We store the `assumption`, `branching`, and `function_return` constraints given by waypoints in the corresponding transitions and they are then used by the predicate CPA at the corresponding program locations. We currently do not support `avoid` waypoints.

Further, we combine the automaton with the existing analyses in CPAchecker. The predicate CPA constructs SMT path formulas T_{stem} and T_{loop} with SSA indices for variables to distinguish different unrollings. The formula T_{stem} encodes the executions of the program leading from initial states to the loop described by the cycle segments, and the formula T_{loop} expresses the executions of this loop. It uses the automaton CPA to make these formulas more strict at the locations described by the automaton. The path formulas are used by the termination CPA to determine whether there exists a state that can be reached twice, following the idea of liveness-to-safety reduction.

The termination CPA unrolls the loop and checks whether there exists a state that can be visited twice by the loop. For every loop head in the program, its abstract state keeps track of how many times the loop was unrolled, and creates special variables that represent all the past values of the variables at the loop head. For example, after exploring paths with two loop unrollings in the program from Fig. 1, the abstract state of this CPA would contain $\{\mathtt{i}' = \mathtt{i}_0 \vee \mathtt{i}' = \mathtt{i}_1\}$. The primed variables are similar to the ghost variables in the liveness-to-safety reduction and represent any previous value of `i` seen at the loop head. The automaton CPA returns a conjunction of all the assumptions given by the witness for the specific unrolling. For example, the automaton constructed for the witness in Fig. 1 gives only the constraint for the stem, which is $\mathtt{i}_0 = 5$. Putting everything together, we obtain the following formula for one unrolling of the witness in Fig. 1:

$$\underbrace{\big(\mathtt{i}_0 > 0 \wedge ((\mathtt{i}_0 \neq 5 \wedge \mathtt{i}_1 = \mathtt{i}_0 - 1) \vee (\mathtt{i}_0 = 5 \wedge \mathtt{i}_1 = \mathtt{i}_0))\big)}_{\text{Predicate CPA}} \wedge$$

$$\wedge \underbrace{\big(\mathtt{i}_0 = \mathtt{i}'\big) \wedge \big(\mathtt{i}_1 = \mathtt{i}'\big)}_{\text{Termination CPA}} \wedge \underbrace{\big(\mathtt{i}_0 = 5\big)}_{\text{Automaton CPA}}$$

If such a formula is satisfiable, there exists an infinite execution and the witness is validated.

THETA. THETA can validate non-termination witnesses by annotating its internal CFA representation of the input program with information from the witness file. Using this annotated CFA, THETA can either use a modular validation technique, or it can use any of its existing model-checking algorithms supporting the termination property on the annotated CFA to validate the witness. Both of these validation methods are new additions to THETA as part of this work, except for their reuse of the existing model-checking analyses.

Annotating the CFA: THETA represents the input programs as control-flow automata (CFA), parsed directly from the C source. *Abstract syntax tree* (AST) nodes are stored as metadata on the edges of the CFA. This enables THETA to find which CFA edges correspond to which waypoint locations in the witness, to add additional assumptions as state-space guards.

A segment counter variable is added to the model, which is incremented whenever a segment's `follow` or `cycle` waypoint has been passed, and reset after the last cycle segment to the number of the first cycle segment. THETA does not yet support `avoid` waypoints. The assumptions added to the CFA state that if the value of the segment counter is currently n and the location corresponds to the waypoint in the n-th segment of the witness, then the constraint of the waypoint must be satisfied. THETA supports the `assumption`, `function_return`, and `branching` waypoint types.

For example, if the second waypoint with type `assumption` contains the constraint $x = 1$, then

$$\texttt{segment\_counter} = 2 \Rightarrow x = 1$$

is added to the edge just before the statement that the location information points to.

1) Modular Validation: THETA implements a cheap, but incomplete check using the annotated CFA. This check can not refute witnesses, but can often validate them successfully.

First, THETA reconstructs the *stem*-trace of the lasso based on the `follow` waypoints in the witness, and encodes it as an SMT formula T_{stem}. Then, it also reconstructs the *loop* of the lasso based on the following `cycle` waypoints, and encodes it as T_{loop}. If there is a program state that is both reachable after T_{stem}, and from which —after T_{loop}— the same state is reachable, then that program state can be used as the recurrent set [35]. The existence of such a state can be checked using a quantifier-free SMT formula. This is a strict check, because the loop has to be unrolled enough times in the witness that T_{loop} can repeat the exact same state after every iteration — e.g., a loop variable repeating the values $\{0,1\}$ periodically would necessitate the loop to be unrolled twice (or a larger multiple of 2).

A less strict (but still incomplete) check assumes the witness encodes a recurrence condition as a constraint in the loop. While detecting if a waypoint was meant to be such a statement is not straightforward, we can assume that all assumptions added after passing the last `follow` waypoint and before the first program statement which changes the value of a variable may serve such a condition. Thus, THETA collects all assumptions in the loop that refer to the first SSA-index in the SMT encoding of T_{loop}, and takes their conjunction, resulting in R. Then, it checks if a state in R is reachable after T_{stem} and if for every state r_1 satisfying R, a successor state r_2 exists that also satisfies R, and which can be reached from r_1 by T_{loop}. The latter check is more expensive than the case of the concrete program state, as we cannot encode this query without quantifiers.

In both cases, THETA can output a (potentially strengthened) non-termination argument in the form of a recurrent set [35]. However, correct witnesses exist that this approach does not validate, and therefore, we do not use the modular validation technique to refute witnesses.

2) Validation Based on Model Checking: For a more generic approach that can also refute witnesses, THETA can utilize all its verification algorithms presented in Section IV on the annotated CFA. Here —in contrast to modular validation— we can not only confirm witnesses that allow THETA to re-establish the non-termination argument, but can also refute witnesses if they include incorrect information about the lasso.

As with verification, THETA uses a loop-checking CEGAR configuration, EMERGENTHETA uses k-induction, and THORN uses a mapping to CHCs for validation as well. THETA-modular uses the modular validation with a direct SMT-encoding.

WITCH. The violation-witness validator WITCH [11] is based on symbolic execution. It uses and extends parts of the SYMBIOTIC framework, most notably the symbolic executor JETKLEE, a fork of KLEE [64] developed for the purposes of SYMBIOTIC. The tool previously supported only witnesses of safety properties, and we adapted it to also support witnesses of non-termination.

As JETKLEE accepts programs in the LLVM intermediate representation, WITCH first instruments the C code according to the witness, adding assumptions and marking branching points. The program is compiled to LLVM IR and the witness is adjusted so that it only uses information preserved by the compilation (see [11] for details).

Each loop in the LLVM IR program is then instrumented following the idea of liveness-to-safety reduction, introducing a special function for the choice whether to store the current values of loop-modified variables, and special assertions for checking that the current values differ from the values stored in some previous iteration. The instrumented program is then passed to the symbolic executor together with the witness.

The general approach to witness validation in WITCH is to synchronously explore the program and the witness, mapping parts of the program executions to witness segments. It assigns a witness segment to each symbolic state, with the initial symbolic state being assigned the first segment. When the execution passes the `follow` waypoint of the assigned segment, WITCH associates the next state with the next witness segment. Throughout the process, it uses the constraints from the witness to reduce the explored state space.

We build upon this approach in our extension for non-termination witnesses. After a program execution passes the `follow` waypoint of the last normal segment, we assign the first `cycle` segment to the next symbolic state. When presented with the choice of storing the current state of loop-modified variables, the symbolic execution forks — on one branch we store the values and remember the current witness segment, and on the other we omit this step to preserve the previously remembered values. However, to be more efficient, the values can be stored in the ghost variables only if the symbolic state is assigned a `cycle` segment. On top of that, if there already was such a non-deterministic choice on the path to this symbolic state, and we stored the values, the values are not stored again. This means that apart from initialization the values are stored at most once on each program execution. Both of these optimizations are done on the side of the symbolic executor.

Upon encountering an assertion comparing the stored and the current values of variables, WITCH first checks that the current witness segment is the same as the one remembered at the time of the storing, and that the `cycle`-action waypoint of this segment has been passed since. Only then the assertion is evaluated. If the assertion fails, there exists an infinite execution of the program that is described by the witness, and the witness is confirmed. If WITCH explores all program paths without confirming the witness, it means that there is no infinite execution described by the witness and the witness is refuted.

The main limitation of this approach is that it handles only loops that return to the same concrete state and modify only program variables, and does not support recursion. Further, unless the witness significantly reduces the state space, it inherits the path-explosion problem of symbolic execution, and, if a program is possibly non-terminating but the witness is incorrect, it can not refute the witness unless some of the visited waypoints restrict the exploration to only finite runs.

VI. EVALUATION

With our evaluation, we aim to answer the following research questions:

RQ 1 (Validation Effectiveness): How many witnesses exported by the new verifiers can the new validators validate?

RQ 2 (Validation Efficiency): Can the information in the witnesses improve the efficiency of the validation compared to the verification?

RQ 3 (Non-trivial Information): Do the newly implemented tools export more non-trivial information in the witnesses than tools of SV-COMP 2025 exporting witnesses 1.0?

Benchmark Set. In our evaluation, we utilize SV-Benchmarks, a large and diverse benchmark set of C programs, in the version used by SV-COMP 2025 [12]. The dataset contains programs with known verdicts for different specifications, including termination. We use all 992 tasks that are known to be non-terminating.

Benchmark Environment. For conducting our evaluation, we use BENCHEXEC in version d33e473 to ensure reliable benchmarking [65] on a cloud cluster orchestrated by BENCHCLOUD [66]. 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 the Ubuntu 24.04 operating system. In our experiments, we use 900 s as time limit and 15 GB as memory limit for verification, and 90 s as time limit and 7 GB as memory limit for validation. These are the same time and memory limits as used in SV-COMP. We restrict all experiments to 2 processing units.

Tool Support. Table I summarizes the current state of the tool support for non-termination witnesses in format 1.0 and in format 2.1. Notably, we have more than doubled the number of available validators supporting the new format compared to format 1.0. Additionally, we have implemented the export of witnesses in the new format for almost half of the verifiers that could export witnesses 1.0.

In our evaluation, we use the verifiers THETA [67], EMERGENTHETA [68], and THORN [47] in version 3caaa21, and SYMBIOTIC [7] in version 9cd025a. In addition we use TRANSVER [9] in version 5fc7ca66 using CPACHECKER in version 66247485 to transform the termination tasks to reachability tasks and construct the non-termination witnesses from the violation witnesses for reachability. We chose UAUTOMIZER [69] in its version from SV-COMP 2025 and CPACHECKER [10] in version b20e3a16 as reachability verifiers, since they ranked first and

TABLE I: Current tool support for non-termination witnesses by active participants of SV-COMP 2025 or their components; the horizontal line divides the tools where we implemented the support of format 2.1 (above) and the tools that support only non-termination witnesses in format 1.0 (below); the symbol ○ denotes that almost all the witnesses 1.0 produced by the respective tool in SV-COMP 2025 were syntactically incorrect, even though some of them were confirmed by some validators

	Export		Validation	
Tool	1.0	2.1	1.0	2.1
CPAchecker	●		●	●
EmergenTheta	○	●		●
Symbiotic	●	●		
Theta	○	●		●
Thorn	○	●		●
TransVer		●		
Witch				●
2ls [70]	●			
APRoVE [33]	○			
Bubaak [34]	○			
Proton [71]	●			
UAutomizer [72]	●		●	

second in the *Overall* category of SV-COMP 2025, respectively. All the tools and their methodology to construct non-termination witnesses are described in Sect. IV. For validation, we use CPAchecker [61] in version b20e3a16, EmergenTheta, Theta, and Thorn in version 3caaa21, and Witch [11] in version b717fec, utilizing the methods described in Sect. V.

A. RQ 1: Validation Effectiveness

One of the main goals of the new witness format is to improve the interoperability of tools that export and verify non-termination arguments. To achieve this goal, we implemented in several tools the export and validation of witnesses in the new format. Table II summarizes the results of validating the exported witnesses. In sum, all tools together produced 1379 witnesses for the 992 tasks.

Symbiotic produced a non-termination witness for 658 (66 %) of the tasks, which is the highest number among our verifiers. Due to the missing support for programs with arrays and dynamic memory allocation, TransVer was able to transform only 88 (8 %) non-termination tasks into reachability tasks. Therefore, CPAchecker and UAutomizer produced only 71 and 73 witnesses, respectively. Configurations of Theta lack support for function pointers and complex dynamic memory structures, therefore, they could not process the majority of the programs (558), leaving 434 potentially verifiable tasks. From this, Thorn produced a witness for 221 (51 %), Theta for 205 (47 %), and EmergenTheta for 151 (35 %) tasks. Together, these configurations produced a witness for 246 (57 %) unique tasks.

All the witnesses produced by Symbiotic and the verification chains using TransVer were validated by at least one validator. Moreover, 143 (95 %), 173 (84 %), and 199 (90 %) of the witnesses produced by EmergenTheta, Theta, and Thorn, respectively, were validated by at least one validator. The fact that almost all the produced witnesses were confirmed by at least one validator indicates their high quality.

We observe that the highest number of validated witnesses per verifier differs, i.e., no validator is the best for all of them. Interestingly, the best validating algorithm is often not from the same framework. For example, CPAchecker outperformed Theta in validating the witnesses produced by Theta. This shows that information exchange among different verification tools is advantageous. In particular, Theta-modular could validate 80 additional tasks using information from the witnesses of other tools.

> These observations indicate that the wide variety of the non-termination arguments produced by different algorithms can be successfully expressed in the proposed witness format extension. Additionally, they show that the format is suitable for exchanging information between different tools.

B. RQ 2: Validation Efficiency

The main goal of verification witnesses is to encode information about the verification process in order to allow for independent validation. One major consequence of this is that the validation, having access to information collected during the verification, can be more efficient than the verification itself. In particular, for counterexamples, their validation should be more efficient than the verification since the witnesses constrain the search space. This is reflected by the lower time-limit for the validation compared to verification, i.e., 90 s vs. 900 s.

For each of the considered verifiers, Fig. 4 shows the time taken by the verifier to produce a witness (y-axis) and the time needed by individual validators to confirm the witness (x-axis). In many cases, the witness can be validated more than 10 times faster than the initial verification of the program.

The results also show the difference between the validation techniques. For Witch, which is based on symbolic execution, the restrictions of the program paths stemming from the witness help reduce the validation time. However, for Thorn and EmergenTheta, the opposite result can be observed: they take longer to validate their own witnesses than to verify the original task. For Thorn, this can be explained by the larger size (and complexity) of the CHC-encoding, which may result in a slower solution inside the CHC-solver. For EmergenTheta, we theorize that in its transition-system-based encoding, control-flow information may be removed during the verification. During the validation, this is no longer possible, due to having to keep track of the segment order in the witness. Therefore, the witness may add additional proof goals in the formulas used for k-induction. However, the witness also has to be checked for potential issues, which adds overhead to the original verification performance. Therefore, these tools as validators underperform themselves as verifiers.

It is apparent in most of the scatter plots that the validators tend to be faster than the verifiers, except Thorn and EmergenTheta, for the reasons already discussed. The only exception is the validation results of Symbiotic witnesses. We can observe that only Witch is able to outperform Symbiotic. The reason is that both are implemented in C++, whereas all the other tools use Java engines that run on the Java Virtual Machine (JVM). The JVM typically takes 5–10 seconds to start. Therefore, subtracting the JVM startup time of the Java-based tools, the validation would be in many cases as fast as the verification or faster.

TABLE II: Numbers of exported non-termination witnesses in format 2.1 and the results of their validation; exported witnesses are below each verifier, and pairs x / y mean that the corresponding validator confirmed x and refuted y of them

Validators	EMERGENTHETA 151 witnesses	THETA 205 witnesses	THORN 221 witnesses	TRANSVER-CPACHECKER 71 witnesses	TRANSVER-UAUTOMIZER 73 witnesses	SYMBIOTIC 658 witnesses
CPACHECKER	115 / 0	**156** / 0	172 / 0	**69** / 0	**66** / 0	204 / 0
EMERGENTHETA	62 / 1	37 / 1	61 / 1	39 / 1	41 / 0	32 / 3
THETA	**131** / 1	109 / 2	**177** / 2	39 / 0	28 / 0	158 / 0
THETA-modular	23 / 0	15 / 0	25 / 0	0 / 0	0 / 0	241 / 0
THORN	56 / 1	36 / 0	61 / 1	39 / 2	37 / 2	43 / 3
WITCH	104 / 0	148 / 0	164 / 0	57 / 0	64 / 0	**658** / 0
Total confirmed	143	173	199	71	73	658

TABLE III: Numbers of exported non-termination witnesses in format 1.0 in SV-COMP 2025; exported witnesses are below each verifier, and pairs x / y mean that the corresponding validator confirmed x and refuted y of them

Validators	2LS 585 witnesses	CPACHECKER 637 witnesses	PROTON 667 witnesses	SYMBIOTIC 623 witnesses	UAUTOMIZER 715 witnesses
CPACHECKER	**385** / 8	486 / 0	**422** / 1	**350** / 1	431 / 41
UAUTOMIZER	181 / 380	**519** / 13	128 / 467	56 / 538	**685** / 2
Total confirmed	487	552	505	353	687

TABLE IV: P-values for a Wilcoxon signed-rank test comparing the CPU-time of validation and verification; missing values (–) are due to the small sample size available for the test; values less than 0.01 printed in green, values greater than 0.9 in red

Validators	EMERGENTHETA	THETA	THORN	TRANSVER-CPACHECKER	TRANSVER-UAUTOMIZER	SYMBIOTIC
CPACHECKER	2.8×10^{-11}	6.0×10^{-24}	2.8×10^{-30}	2.6×10^{-13}	6.8×10^{-12}	1.0
EMERGENTHETA	1.0	1.0	7.6×10^{-11}	8.1×10^{-10}	4.5×10^{-13}	1.0
THETA	2.3×10^{-12}	1.0	1.1×10^{-30}	1.8×10^{-12}	3.7×10^{-9}	1.0
THETA-modular	2.5×10^{-5}	0.95	3.0×10^{-8}	–	–	1.0
THORN	1.0	1.0	1.0	1.0	3.1×10^{-3}	1.0
WITCH	4.3×10^{-19}	2.5×10^{-26}	5.8×10^{-29}	2.6×10^{-11}	1.8×10^{-12}	1.9×10^{-108}

Wilcoxon Signed-Rank Test. Since it is often difficult to correctly infer the density of data points clustered in the same region of scatter plots, we additionally report the p-values of the *Wilcoxon Signed-Rank Test* [73] for each verifier-validator pair in Table IV. The null hypothesis of the test states that the distribution of differences between verification and validation times is centered around zero – that is, the two are approximately equal. The alternative hypothesis posits that verification time is stochastically greater than validation time. The reported p-values represent the probability of incorrectly rejecting the null hypothesis. Therefore, lower p-values indicate higher confidence that validation is, on average, faster than verification. The statistical results confirm the conclusions previously suggested by the scatter plots.

The results of our experiments mostly confirm that the non-termination witnesses contain useful information that helps rediscover non-terminating executions in less time.

C. RQ 3: Non-trivial Information

Precisely defining and measuring quality of witnesses, i.e., how useful the included information is, is a difficult task. However, the goal of our work is to add support for exporting non-termination witnesses for multiple verifiers in a manner that they should not export trivial witnesses. Thus, we compare the witness quality between our results and the results of the state-of-the-art tools of the witness 1.0 format in SV-COMP 2025.

All tools participating in SV-COMP 2025 were required to export witnesses for non-termination in version 1.0. Some tools circumvent this by exporting minimal witnesses or reachability witnesses that do not include invariants and cycle heads. Although a reachability witness can still provide useful information regarding the stem, the presence of a target state and the lack of information regarding the cycle should render it invalid. Unfortunately, such witnesses were still validated by some validators, thus the actual number of tools exporting proper non-termination witnesses is not apparent at first glance.

To count and compare non-trivial witnesses, we collected all non-termination violation witnesses 1.0 that were produced in SV-COMP 2025 [74] and filtered out witnesses which did not contain a *cyclehead*, or the tool's answer was not *false*, or the expected verdict was *true*, or they were not approved by the witness linter.

Overall, we found 5 (active, non-meta) tools out of 12 that produced valid, non-trivial witnesses and 2 validators capable of validating a subset of these witnesses. The results are shown in Table III, while similar results for our new witness-format experiment were shown in Table II.

The differing number of tools makes the comparison more difficult, but we are comparing a fairly large number of non-termination witnesses, so we can still draw some conclusions on proportions based on the data. We collected 3227 witnesses 1.0 and 1347 witnesses 2.1, of which 1559 (48 %) and 1272 (94 %) were validated by at least one tool and refuted by none, respectively. Furthermore, the number of refuted witnesses is much lower for the witness 2.1 format, as shown in Table III.

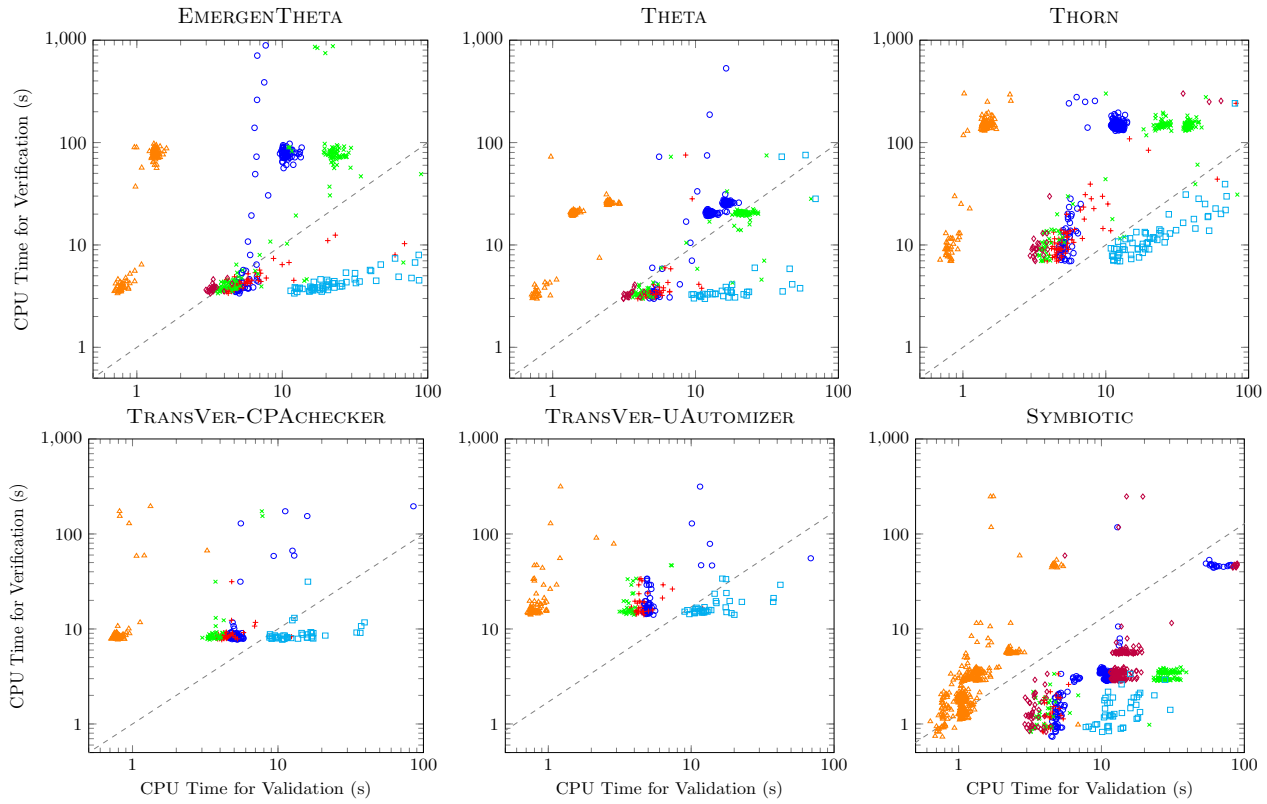

Fig. 4: Scatter plots comparing the time for verification and validation for validators CPAchecker (○), EmergenTheta (+), Theta (×), Theta-modular (◇), Thorn (□), and Witch (△); $x = y$ drawn as gray and dashed line; producing verifier above plot

Although the total number of collected witnesses differs significantly, the high validation rate and lower refutation rate for witness in format 2.1 suggest that it offers a significant improvement over format 1.0.

Based on these results, we believe that broader adoption of the format can significantly enhance the quality of the exported information in non-termination witnesses.

D. Threats to Validity

Internal Validity. We ensured consistency and accuracy of our experiments using the BenchExec framework [65], which relies on modern GNU/Linux features for reliable benchmarking. CPU time —and thus task counts— may still be affected by external factors like temperature fluctuations or tool non-determinism. Bugs in tools are also possible, but using multiple independent validators minimizes their impact.

External Validity. Our results may not generalize well due to the limited size and diversity of the benchmark set. Since SV-COMP [12] uses the same benchmark set, tools might be tuned specifically towards the benchmark set, limiting generalization to other tasks. Also, only a subset of model-checking algorithms were represented by the evaluated tools, so results may not extend to all approaches. However, the diversity of tools in this paper helps to reduce this risk.

Construct Validity. The *CPU time* and *solved tasks* measures are the same used in SV-COMP [12], and thus known to be accepted by the software-verification community. Only known non-terminating tasks from SV-Benchmarks were included, so refutation capabilities of validators were not assessed.

VII. Conclusion

We have introduced an extension of the format for software-verification witnesses to support non-termination witnesses. We extended several state-of-the-art open-source tools for software verification and witness validation to export and validate non-termination witnesses in the extended format using various approaches. Our experimental evaluation shows that the developed tools support producing non-termination arguments that can be validated by independent tools. We included our format extension in the witness format 2.1 [75], together with other extensions [14, 15]. The new format, the validation benchmarks, and the extended validators are offered to validate non-termination witnesses in SV-COMP 2026 and beyond.

Data-Availability Statement. Our reproduction package [76], which includes all software and data that we used for our experiments, and the set of example witnesses [13] to demonstrate how the new format works, are available on Zenodo. Interactive tables of the experimental results are available at our supplementary web page: https://www.sosy-lab.org/research/non-termination-witnesses/.

Funding Statement. Zs. Ádám and L. Bajczi were supported by projects EKOP-24-3-BME-{288,213}, DKÖP 400433/2023, DKÖP 400434/2023, Erasmus, BAYHOST MobFA2025/15, and BAYHOST MobFA2025/16. P. Ayaziová and J. Strejček were supported by the Czech Science Foundation grant GA23-06506S and by the Bayerisch-Tschechische Hochschulagentur BTHA-MOB-2025-7 and BTHA-MOB-2025-9. D. Beyer, M. Jankola, and M. Lingsch-Rosenfeld 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). doi:10.1145/2786805.2786867

[2] Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). doi:10.1007/978-3-662-46681-0_31

[3] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). doi:10.1145/2950290.2950351

[4] 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). doi:10.1145/3477579

[5] Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. pp. 184–203. LNCS 14624, Springer (2024). doi:10.1007/978-3-031-66149-5_11

[6] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). doi:10.1007/978-3-031-57256-2_15

[7] 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 (3). pp. 406–411. LNCS 14572, Springer (2024). doi:10.1007/978-3-031-57256-2_29

[8] Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Zs., Dobos-Kovács, M., Szekeres, D., Mondok, M., Molnár, V.: Theta: Abstraction based techniques for verifying concurrency (competition contribution). In: Proc. TACAS (3). pp. 412–417. LNCS 14572, Springer (2024). doi:10.1007/978-3-031-57256-2_30

[9] Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Xia, T., Zheng, X.: TransVer: A modular program-transformation framework for reduction to reachability. In: Proc. SPIN. LNCS 15945, Springer (2025). doi:10.1007/978-3-032-06847-7_1

[10] Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). doi:10.1007/978-3-031-71177-0_30

[11] Ayaziová, P., Strejček, J.: Witch 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Proc. TACAS (3). pp. 341–346. LNCS 14572, Springer (2024). doi:10.1007/978-3-031-57256-2_18

[12] Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_9

[13] Ádám, Zs., Ayaziová, P., Bajczi, L., Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Strejček, J.: Non-termination witnesses in format 2.1 for ASE 2025 article 'Non-termination witnesses and their validation'. Zenodo (2025). doi:10.5281/zenodo.17264555

[14] Erhard, J., Bentele, M., Heizmann, M., Klumpp, D., Saan, S., Schüssele, F., Schwarz, M., Seidl, H., Tilscher, S., Vojdani, V.: Correctness witnesses for concurrent programs: Bridging the semantic divide with ghosts. In: Proc. VMCAI, Part I. pp. 74–100. LNCS 15529, Springer (2025). doi:10.1007/978-3-031-82700-6_4

[15] Heizmann, M., Klumpp, D., Lingsch-Rosenfeld, M., Schüssele, F.: Correctness witnesses with function contracts. arXiv/CoRR **2501**(12313) (January 2025). doi:10.48550/arXiv.2501.12313

[16] McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). doi:10.1016/j.cosrev.2010.09.009

[17] Beyer, D., Wendler, P.: Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In: Proc. SPIN. pp. 1–17. LNCS 7976, Springer (2013). doi:10.1007/978-3-642-39176-7_1

[18] Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). doi:10.1007/978-3-030-61362-4_26

[19] 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). doi:10.1007/978-3-319-92994-1_1

[20] 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). doi:10.1007/978-3-319-41540-6_28

[21] Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). doi:10.1007/978-3-319-21401-6_6

[22] Sternagel, C., Thiemann, R.: The certification problem format. In: Proc. UITP. pp. 61–72. EPTCS 167, EPTCS (2014). doi:10.4204/EPTCS.167.8

[23] Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. In: Proc. LMS. vol. s2-42, pp. 230–265. London Mathematical Society (1937). doi:10.1112/plms/s2-42.1.230

[24] Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT Numerical Mathematics **8**, 174–186 (1968). doi:10.1007/BF01933419

[25] Manna, Z.: Termination of programs represented as interpreted graphs. In: Spring Joint Computer Conf. pp. 83–89. ACM (1970). doi:10.1145/1476936.1476956

[26] Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Proc. VMCAI. pp. 239–251. LNCS 2937, Springer (2004). doi:10.1007/978-3-540-24622-0_20

[27] Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: Proc. POPL. pp. 132–144. ACM (2005). doi:10.1145/1040305.1040317

[28] Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Proc. CAV. pp. 415–418. LNCS 4144, Springer (2006). doi:10.1007/11817963_37

[29] Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. PLDI. pp. 415–426. ACM (2006). doi:10.1145/1133981.1134029

[30] Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electr. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006). doi:10.1016/j.entcs.2005.11.018

[31] Metta, R., Karmarkar, H., Madhukar, K., Venkatesh, R., Chakraborty, S.: Proton: Probes for non-termination and termination (competition contribution). In: Proc. TACAS (3). pp. 393–398. LNCS 14572, Springer (2024). doi:10.1007/978-3-031-57256-2_27

[32] Metta, R., Yeduru, P., Karmarkar, H., Medicherla, R.K.: VeriFuzz 1.4: Checking for (non-)termination (competition contribution). In: Proc. TACAS (2). pp. 594–599. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_42

[33] Lommen, N., Giesl, J.: AProVE (KoAT + LoAT) (competition contribution). In: Proc. TACAS (3). pp. 205–211. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_13

[34] Chalupa, M., Richter, C.: Bubaak: Dynamic cooperative verification (competition contribution). In: Proc. TACAS (3). pp. 212–216. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_14

[35] Gupta, A.K., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.: Proving non-termination. In: Proc. POPL. pp. 147–158. ACM (2008). doi:10.1145/1328438.1328459

[36] Han, Z., He, F.: Data-driven recurrent set learning for non-termination analysis. In: Proc. ICSE. pp. 1303–1315. IEEE (2023). doi:10.1109/ICSE48619.2023.00115

[37] Bakhirkin, A., Piterman, N.: Finding recurrent sets with backward analysis and trace partitioning. In: Proc. TACAS. pp. 17–35. Springer (2016). doi:10.1007/978-3-662-49674-9_2

[38] Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: Proc. FMCAD. pp. 221–230. IEEE (2019). doi:10.23919/FMCAD.2019.8894271

[39] Frohn, F., Fuhs, C.: A calculus for modular loop acceleration and non-termination proofs. Int. J. Softw. Tools Technol. Transfer **24**(5), 691–715 (October 2022). doi:10.1007/s10009-022-00670-2

[40] Chatterjee, K., Goharshady, E.K., Novotný, P., Žikelić, D.: Proving non-termination by program reversal. In: Proc. PLDI. pp. 1033–1048. ACM (2021). doi:10.1145/3453483.3454093

[41] Chen, H.Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Proc. TACAS. pp. 156–171. Springer (2014). doi:10.1007/978-3-642-54862-8_11

[42] Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for Java bytecode. In: Formal Verification of Object-Oriented Software. pp. 123–141. Springer (2012). doi:10.1007/978-3-642-31762-0_9

[43] Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In: Proc. TAP. pp. 154–170. Springer (2008). doi:10.1007/978-3-540-79124-9_11

[44] Chalupa, M., Jašek, T., Tomovič, L., Hruška, M., Šoková, V., Ayaziová, P., Strejček, J., Vojnar, T.: SYMBIOTIC 7: Integration of PREDATOR and more (competition contribution). In: Proc. TACAS (2). pp. 413–417. LNCS 12079, Springer (2020). doi:10.1007/978-3-030-45237-7_31

[45] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The Program Dependence Graph and Its Use in Optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987). doi:10.1145/24039.24041

[46] Chalupa, M., Klaška, D., Strejček, J., Tomovič, L.: Fast computation of strong control dependencies. In: Proc. CAV. LNCS, vol. 12760, pp. 887–910. Springer (2021). doi:10.1007/978-3-030-81688-9_41

[47] Hajdu, Á., Micskei, Z.: Efficient strategies for CEGAR-based model checking. J. Autom. Reasoning **64**(6), 1051–1091 (2020). doi:10.1007/s10817-019-09535-x

[48] Bajczi, L., Ádám, Zs., Molnár, V.: C for Yourself: Comparison of Front-End Techniques for Formal Verification. In: Proc. FormaliSE. IEEE (2022). doi:10.1145/3524482.3527646

[49] Ádám, Zs., Bajczi, L., Dobos-Kovács, M., Hajdu, A., Molnár, V.: THETA: Portfolio of CEGAR-based analyses with dynamic algorithm selection (competition contribution). In: Proc. TACAS (2). pp. 474–478. LNCS 13244, Springer (2022). doi:10.1007/978-3-030-99527-0_34

[50] Mondok, M., Vörös, A.: Abstraction-based model checking of linear temporal properties. In: Proc. PhD Mini-Symposium. pp. 29–32. Budapest University of Technology and Economics, Department of Measurement and Information Systems (2020), available at http://real.mtak.hu/id/eprint/107359

[51] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). doi:10.1007/10722167_15

[52] Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: Proc. FMCAD. pp. 1–7. IEEE (2018). doi:10.23919/FMCAD.2018.8603013

[53] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003). doi:10.1016/S0065-2458(03)58003-2

[54] Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS. pp. 351–368. LNCS 6887, Springer (2011). doi:10.1007/978-3-642-23702-7_26

[55] Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning **69**(1), 5 (2025). doi:10.1007/s10817-024-09702-9

[56] Cimatti, A., Griggio, A.: Software model checking via IC3. In: Proc. CAV. pp. 277–293. LNCS 7358, Springer (2012). doi:10.1007/978-3-642-31424-7_23

[57] Mondok, M., Molnár, V.: Efficient Manipulation of Logical Formulas as Decision Diagrams. In: Proc. PhD Mini-Symposium. pp. 61–65. Budapest University of Technology and Economics, Department of Measurement and Information Systems (2024). doi:10.3311/MINISY2024-012

[58] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SEAHORN verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). doi:10.1007/978-3-319-21690-4_20

[59] Ernst, G.: KORN: Horn clause based verification of C programs (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_36

[60] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). doi:10.1109/FMCAD.2009.5351147

[61] Beyer, D., Lingsch-Rosenfeld, M.: CPACHECKER 4.0 as witness validator (competition contribution). In: Proc. TACAS (3). pp. 192–198. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_11

[62] Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). doi:10.1007/978-3-540-73368-3_51

[63] Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). doi:10.1007/s10817-017-9432-6

[64] Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008), available at https://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[65] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). doi:10.1007/s10009-017-0469-y

[66] Beyer, D., Chien, P.C., Jankola, M.: BENCHCLOUD: A platform for scalable performance benchmarking. In: Proc. ASE. pp. 2386–2389. ACM (2024). doi:10.1145/3691620.3695358

[67] Telbisz, C., Bajczi, L., Szekeres, D., Vörös, A.: THETA: Various approaches for concurrent program verification (competition contribution). In: Proc. TACAS (3). pp. 260–265. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_22

[68] Mondok, M., Bajczi, L., Szekeres, D., Molnár, V.: EMERGENTHETA: Variations on symbolic transition systems (competition contribution). In: Proc. TACAS (3). pp. 217–222. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_15

[69] 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 (3). pp. 418–423. LNCS 14572, Springer (2024). doi:10.1007/978-3-031-57256-2_31

[70] Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and loop unwinding (competition contribution). In: Proc. TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_31

[71] Mukhopadhyay, D., Metta, R., Karmarkar, H., Madhukar, K.: PROTON 2.1: Synthesizing ranking functions via fine-tuned locally hosted LLM (competition contribution). In: Proc. TACAS (3). pp. 242–247. LNCS 15698, Springer (2025). doi:10.1007/978-3-031-90660-2_19

[72] Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER 2023 (competition contribution). In: Proc. TACAS (2). pp. 577–581. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_39

[73] Woolson, R.: Wilcoxon Signed-Rank Test, pp. 1–3. John Wiley & Sons, Ltd (2008). doi:10.1002/9780471462422.eoct979

[74] Beyer, D., Strejček, J.: Verification witnesses from verification tools (SV-COMP 2025). Zenodo (2025). doi:10.5281/zenodo.15012077

[75] Beyer, D., Strejček, J.: SV-Witnesses – Format 2.1. Zenodo (2025). doi:10.5281/zenodo.17277275

[76] Ádám, Zs., Ayaziová, P., Bajczi, L., Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Strejček, J.: Reproduction package for ASE 2025 proceedings 'Non-termination witnesses and their validation'. Zenodo (2025). doi:10.5281/zenodo.17237968