Transition Invariants Revisited: Termination Witnesses and Their Validation

Dirk Beyer, Marek Jankola, and Marian Lingsch-Rosenfeld

LMU Munich, Munich, Germany

Abstract. Whenever automated provers such as automatic software verifiers deliver a verdict (true or false), they are expected to produce also a witness that justifies the verdict. This allows independent validation of the verdict using the witness by a third party, increasing trust in the results. The current standard exchange formats for witnesses in software verification do not support program termination. To fill this gap, we propose an extension of the witness format that is based on *transition invariants* as a *general* and *effective* formalism. We justify this by (a) proving that transition invariants can encode other popular termination arguments like ranking functions and (b) providing three different validation approaches for transition invariants, which together can validate most of the exported witnesses. Our approach based on transition invariants was integrated into version 2.1 of the recently released witness format, and the software-verification community has adopted the format for SV-COMP.

Keywords: Formal Verification · Software Model Checking · Witness Validation · Termination · Reachability · Invariants · Software Verification · Program Analysis

1 Introduction

Given a program P and a specification φ, the goal of software verification is to determine whether the program satisfies the specification, denoted as $P \models \varphi$ (cf. [1–4]). In this paper, we focus on proving and validating program termination. The termination property requires that a program is free of infinite cycles (we do not consider other reasons for non-termination, such as deadlocks and blocked functions, as these can be checked by reachability queries). Proving program termination has a significant relevance in industrial practice [5–7].

While verification enhances the quality of software, automatic verifiers are complex systems that can have bugs, or can use imprecise approaches such as machine learning [8]. Therefore, verification tools should produce verification witnesses [9–11] that can be independently validated. Although verification witnesses are used since a long time, for example in the competition on software verification (SV-COMP) since 2015 [12], the termination arguments were unfortunately not supported until now by any of the available witness formats.

We use *transition invariants*, proposed more than 20 years ago in the seminal paper by Podelski and Rybalchenko [13], as the unifying concept for expressing all termination arguments. In this paper, we present (1) an *exchange format* for termination witnesses based on transition invariants, (2) the insight that transition

To appear in Proc. CAV 2026

Table 1: Tools with positive score in termination category of SV-COMP

Verifiers	2LS [15, 16]	APROVE [17]	BUBAAK [18, 19]	BUBAAK-SPLIT [20]	CBMC [21]	CPACHECKER [22, 23]	CPV [24]	EMERGENTHETA [25, 26]	ESBMC [27, 28]	GOBLINT [29, 30]	MOPSA [31]	MUVAL	NAGPA [32]	PINAKA [33]	PROTON [34, 35]	REFUNCTION	SYMBIOTIC [36, 37]	THETA [38, 39]	THORN	UAUTOMIZER [40, 41]
SV-COMP 2025																				
Participation	✓	✓	✓	✓		✓		✓	✓	✓			✓		✓		✓	✓	✓	✓
Termination Witnesses																				
SV-COMP 2026																				
Participation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Termination Witnesses						✓	✓	✓	✓	✓	✓	✓	✓				✓	✓	✓	

invariants are *general* enough to encode the existing termination arguments, and (3) three validation approaches that are effective and efficient. To demonstrate the generality of transition invariants, we provide a constructive proof showing how lexicographic ranking functions and safety invariants —synthesized via techniques for reduction from termination to reachability— can be transformed into transition invariants. Furthermore, we present three approaches for validation, based on a reduction to validation of reachability witnesses (Sect. 5.1), proving the existence of a decreasing implicit rank of the transition invariant (Sect. 5.2.1), and verification of termination of a simplified program (Sect. 5.2.2).

We evaluate our approaches by exporting witnesses from multiple verifiers, one that synthesizes ranking functions and three others that perform reductions from termination to reachability. The exported witnesses are subsequently validated using the three proposed validation approaches.

Contributions. We make the following contributions:

- extending the established witness format for software verification to support transition invariants expressing termination arguments (Sect. 3), which has been adopted in version 2.1 [14] of the format,
- showing that existing termination arguments can be expressed as transition invariants (Sect. 4),
- implementing the export of termination witnesses from multiple tools based on the synthesis of ranking functions and a reduction to reachability (Sect. 4),
- presenting and implementing three different approaches for the validation of the termination witnesses and proving their correctness (Sect. 5), and
- conducting an evaluation on the largest known benchmark set for software verification (Sect. 6).

Related Work. Our work builds on, and extends, the results from two research fields, termination analysis and verification witnesses.

Termination Analysis. There are many tools for software verification that support proving termination. For example, a total of 14 verifiers participated in SV-COMP 2025 [42] and received a positive score in the termination category; we list all these tools in Table 1.

When computing termination proofs, verifiers typically synthesize one of the following termination arguments: (1) *(lexicographic) ranking functions* [15, 43–45], (2) *invariants* for programs resulting from a termination-to-safety reduction [22, 35, 46–50], and (3) *transition invariants* [6, 51]. While algorithms exist to validate some of the arguments, for example, for ranking functions [52–58] and k-inductive invariants [49], none of them can validate all the kinds of arguments, and none use a tool-independent exchange format.

Verification Witnesses. Independent validation of verification results is crucial for ensuring correctness of the results. Therefore, research teams have developed various machine-readable formats across different problem domains to express the justification of verification results, such as software [9–11] and hardware verification [59], SAT [60] and SMT [61] solving, and term-rewrite systems [62].

These formats have been adopted quickly by the corresponding communities and their competitions. For example, the software-verification competition (SV-COMP) [42] requires the reachability, no-overflow and non-termination results to be justified by verification witnesses [9–11] and the hardware model-checking competition (HWMCC) [63] requires the violation arguments to be encoded in Btor2 [59] and correctness arguments as circuits [64]. The SAT competition (SAT-COMP) [65] requires the unsatisfiability to be encoded in DRAT [60], which can be mechanically checked using DRAT-trim [66], and the SMT competition (SMT-COMP) [67] requires the unsatisfiability to be encoded as unsat cores [61]. The termination competition (termCOMP) [68] specifies a certification format CPF [62] for termination of first-order term-rewrite systems (TRS).

While termination analysis is an important problem to solve, there is no common exchange format for termination witnesses yet in the area of automatic software verification. Our goal is to close this gap.

2 Preliminaries and Background

To formally reason about the construction of transition invariants and the correctness of the validation approaches, we first introduce our program model. Then, we define the reduction from termination to a reachability problem [47] on our model. Afterward, we formalize standard reachability and termination arguments.

2.1 Program Representation

A *program* $P = (X, S, Init, R)$ consists of

- a finite set X of variables, with a special variable pc that is mapped to the finite domain of ordered program locations,
- a set S of states, i.e., $S = \{s : X \to \mathcal{D} \mid \forall x \in X : s(x) \in \mathcal{D}_x\}$, where $\mathcal{D}_x$ is a set called *domain* of x and $\mathcal{D} = \cup_{x \in X} \mathcal{D}_x$,
- a predicate $Init : S \to \mathbb{B}$, which is true for every initial state, and
- a *transition formula* (or *transition relation*) $R : S \times S \to \mathbb{B}$, which is true if there is a transition between states $s, s' \in S$.

```c
int main(void) {
    int i, j;
    j = 1;
    i = 10000;
    while (i-j >= 1) {
        j = j + 1;
        i = i - 1;
    }
    return 0;
}
```

(a) Running Example

```c
int main(void) {
    int saved = 0; int pc = 0;
    int i1, j1;
    int i, j;
    j = 1;
    i = 10000;
    while (i-j >= 1) {
        if(saved == 0) {pc = 0; i1 = i; j1 = j;}
        if(nondet() && saved == 0)  {
          saved = 1; pc = 1; i1 = i; j1 = j;
        } else {
          assert(saved == 0 || pc != 1
            || i1 != i || j1 != j);
        }
        j = j + 1;
        i = i - 1;
    }
    return 0;
}
```

(b) Instrumented Version

Fig. 1: Program genady.c from SV-Benchmarks (left), instrumented version (right)

Without losing generality, we often refer to predicates over states as sets or relations of states. A *program trace* is a sequence $\langle s_0,\dots,s_m\rangle$ of states, with $\forall 0\leq i<m : R(s_i,s_{i+1})$. A *program loop* is a set of program locations $L=\{l_0,\dots,l_n\}$ for which there exists a program trace $\langle s_0,\dots,s_m\rangle$ with $\forall 0\leq i\leq m : s_i(pc)\in L$ and $s_0(pc)=s_m(pc)$. The *loop head* of a program loop is the smallest program location in the loop and usually corresponds to the beginning of a `while` or `for` statement. We denote the set of all loop heads by L_H. Let $R_L\subseteq S\times S$ be a transition relation of the loop L with $R_L(s,s')$ if s can reach s' after one iteration of the loop L. Furthermore, we assume that if $Init(s)$, then $s(pc)\notin L_H$. Let $X=\{x_0,\dots,x_n\}$ and S be the set of states over X. Consider a predicate $\varphi : S\to\mathbb{B}$. To make the values assigned to variables in the predicate more explicit, we often write $\varphi(s(x_0),\dots,s(x_n))$ instead of $\varphi(s)$. Lastly, let $X'\subseteq X$, then $s_{\downarrow X'} : X'\to\mathcal{D}$ is the projection map of s onto X', with $\forall x\in X' : s(x)=s_{\downarrow X'}(x)$.

Example 1. The program in Fig. 1a has one program loop $L=\{l_5,l_6,l_7\}$ with loop head l_5. The transfer relation of L is

$$R_L(s,s') \text{ iff } s(pc)=s'(pc)=l_5 \wedge s(i)-s(j)\geq 1 \wedge s'(j)=s(j)+1 \wedge s'(i)=s(i)-1$$

2.2 Reduction of Termination to Reachability

A state s' is *reachable* if there is a program trace $\langle s,\dots,s'\rangle$ with $Init(s)$. The set $\mathscr{R}\subseteq S$ is the set of all reachable states. Given a program $P=(X,S,Init,R)$ and a predicate φ over the states S, we say P *satisfies* φ, denoted $P\models\varphi$, if there is no reachable state s such that $\neg\varphi(s)$ holds. The predicate φ is called a *safety property*. We use the terms *safety* and *reachability* interchangeably. The *reachability problem* for program P and predicate φ is to decide whether $P\models\varphi$ holds.

A program P is *terminating* if there is no infinite program trace $\langle s_0,\dots\rangle$ with $Init(s_0)$, otherwise, the program is *non-terminating*. The *termination problem* for program P is to decide whether P is terminating.

A. Biere, C. Artho, and V. Schuppan [47] presented a liveness-to-safety reduction for transition systems with a finite state space. Following their idea, we

reduce termination to a reachability problem for programs with a finite state space. The reduction transforms a given program P into an instrumented program P' and constructs a corresponding safety property φ', such that P terminates if and only if $P' \models \varphi'$. The new program P' contains a ghost variable for every variable of P, and chooses non-deterministically at the beginning of every loop to save the current state into the ghost variables. The safety property expresses that if a state was saved in the ghost variables, then every further visited state must be different. In case a state is revisited, the safety property is violated, and thus the program is non-terminating. Formally, we define the reduction as follows:

Definition 1 (Reduction). *Let $P = (X, S, Init, R)$ be a program with the set of loop heads L_H, the instrumented program P' is the tuple $(X', S', Init', R')$, where*

- $X' = X \cup \widehat{X} \cup \{saved\}$, *where* $\widehat{X} = \{\widehat{x} \mid x \in X\}$,
- $S' = \{s : X' \to \mathcal{D}\}$, *with* $\forall x \in X : \mathcal{D}_{\widehat{x}} = \mathcal{D}_x$ *and* $\mathcal{D}_{saved} = \{0,1\}$,
- $Init'(s) = Init(s_{\downarrow X}) \wedge s(saved) = 0 \wedge \bigwedge_{x \in X} (s(\widehat{x}) = s(x))$,
- $R'(s,s') = R(s_{\downarrow X}, s'_{\downarrow X}) \wedge$

$$\big(\big(\bigwedge_{x \in X} s'(\widehat{x}) = s(\widehat{x})\big) \vee \tag{1}$$

$$(s(saved) = 0 \wedge s'(saved) = 1 \wedge \bigwedge_{x \in X} s'(\widehat{x}) = s(x))\big), \tag{2}$$

with the safety property $\varphi'(s) \equiv (s(pc) \notin L_H \vee s(saved) = 0 \vee \bigvee_{x \in X} s(x) \neq s(\widehat{x}))$.

A transition of type (2) from state s is called *saving* of state $s_{\downarrow X}$ into the ghost variables. Figure 1b displays the instrumented program P'. We add an auxiliary variable `pc` to P' to monitor the program location for the loops (each loop has a unique index), elsewhere, the property is trivially satisfied. Without loss of generality, we restrict saving to loop heads.

2.3 Safety Invariants

Invariants are a standard way to justify the verdict for a safety property. Invariants represent a set of states that overapproximates the set of all reachable states while not containing states that violate the property.

Definition 2. *Let $R(s,s')$ be a transition relation, $\mathscr{R} \subseteq S$ be the set of all reachable states of a program P, and φ a safety property, then formula I is called*

- ***invariant*** *of P if $\forall s \in \mathscr{R} : I(s)$,*
- ***1-step invariant*** *of P if all states $s \in \mathscr{R}$ satisfy $(\exists s' \in \mathscr{R} : R(s',s)) \Rightarrow I(s)$,*
- ***inductive*** *if $\forall s,s' \in S : I(s) \wedge R(s,s') \Rightarrow I(s')$, and*
- ***safe*** *if $I(s) \Rightarrow \varphi(s)$, for every $s \in S$.*

We analogously define invariants for a loop L with relation $R_L(s,s')$ instead of R, overapproximating all the states reachable at the loop head. In the following chapter, we are interested in invariants overapproximating states that are reachable in at least one step. Approaches that produce invariants as proofs for the safety properties are unaffected as every invariant is also a 1-step invariant. We need this slightly weaker condition so that we can later relate the safety invariants to transition invariants.

2.4 Transition Invariants

Definition 3 (Transition Invariant). [13] *Let R be transition relation of a program P, we call relation $T \subseteq S \times S$ a* transition invariant *of P if $R^+ \cap (\mathscr{R} \times \mathscr{R}) \subseteq T$, where R^+ is the transitive closure of R. Furthermore, we call T a* weak transition invariant *if it satisfies a weaker condition $R \cap (\mathscr{R} \times \mathscr{R}) \subseteq T$.*

The relation $T \subseteq S \times S$ is *well-founded* if there is no infinite sequence $s_0, s_1, s_2, \ldots$, such that $\forall i \geq 0 : T(s_i, s_{i+1})$. Furthermore, the relation T is *disjunctively well-founded* if there exist well-founded relations $T_1, \ldots, T_n$, such that $T = T_1 \cup \ldots \cup T_n$. Again, we define a transition invariant for a loop L similarly, using the relation R_L. Furthermore, relation T is *inductive* [13], if $\big(R \cap (\mathscr{R} \times \mathscr{R})\big) \cup \big(T \circ R \cap (\mathscr{R} \times \mathscr{R})\big) \subseteq T$.

Theorem 1. [13] *A program P is terminating if and only if there exists a **disjunctively** well-founded transition invariant T.*

Theorem 2. [13] *A program P is terminating if and only if there exists a well-founded **weak** transition invariant T.*

3 Witnesses for Termination

To allow tools to export transition invariants as witnesses for program termination, we integrated them into the established exchange format for such information: software verification witnesses [14]. We extended version 2.0 of the format [11] by including (loop) transition invariants as a new type of invariant. The extended format with the transition invariants is referred to as version 2.1 of the witness format [14]. Our extension was accepted for SV-COMP 2026 in a demo category. There were 11 verifiers that produced witnesses in our extended format and 4 validators that validated the witnesses.[1] We list all these tools in Table 1, and describe the extension of the format in more detail in Appendix B.

A witness for termination consists of a set of loop, and location (transition) invariants. The location, and loop transition invariants encode the termination arguments, while the location, and loop safety invariants are used to overapproximate the relation space $\mathscr{R} \times \mathscr{R}$. We further denote the set of supporting safety invariants for loop L as $\{I_{L,1}, \ldots, I_{L,k}\}$. For C programs, there are (un-)structured loops, in addition to recursive functions. For structured loops, we use loop transition invariants, while for unstructured loops and recursive functions, we use location transition invariants to express the termination arguments.

In contrast to normal invariants, transition invariants can relate the values of variables in an arbitrary previous state to the values in the current state. For this purpose, we introduce the function `\at(var, AnyPrev)` to refer to the value of `var` in that state. This function can be used in the same way as variables inside of the invariant expressions. Each transition invariant is related to a program location and it must be valid for every pair of reachable states s, s' at that program location such that s' is reachable from s by a program execution. A

[1] https://sv-comp.sosy-lab.org/2026/results/results-validated/

```c
int main() {
  int x = nondet();
  int y = nondet();

  while (x > 0) {
    y = 0;
    while (y < x) {
        y = y + 1;
    }
    x = x - 1;
  }
}
```

```yaml
- entry_type: invariant_set
  metadata: ...
  content:
  - invariant:
      type: transition_loop_invariant
      location:
        line: 5
      value: |
        "(y <= 1 && x <= 0) ||
         (x < \at(x, AnyPrev))"
      format: ext_c_expression
  - invariant:
      type: transition_loop_invariant
      location:
        line: 7
      value: |
        "(x <= \at(x, AnyPrev) &&
            \at(y, AnyPrev) + 1 <= y) ||
         (x + 1 <= \at(x, AnyPrev) && 1 <= x)"
      format: ext_c_expression
```

Fig. 2: Program (left) and a correctness witness with transition invariants (right)

termination witness is *valid* if there exist valid disjunctively well-founded transition invariants for each loop in the program. The transition relation outside of loops is trivially well-founded. Figure 2 shows an example program and a witness containing transition invariants for its two loops.

4 Termination Arguments as Transition Invariants

Defining a format for software verifiers that prove termination with various arguments, requires a formalism that is *general* enough to express all the other formalisms. We can group the existing verification approaches for program termination into three groups, based on the type of termination argument that they synthesize: (a) those constructing transition invariants [6, 51], (b) algorithms synthesizing ranking functions [15, 43–45], and (c) approaches transforming termination into reachability [22, 35, 46–50]. We first establish a stricter property of transition invariants that can be used to validate whether a given transition invariant is a valid termination argument. Afterwards, we present constructive methods to express the other two formalisms as transition invariants.

4.1 Transition Invariants for Termination

A disjunctively well-founded transition invariant for a program P with transition relation R provides an argument for the termination of P. Given a relation induced by a formula T, it is sufficient to check that (a) $R^+ \cap (\mathscr{R} \times \mathscr{R}) \subseteq T$ and (b) T is disjunctively well-founded to show that P always terminates. For our purposes, we use stronger requirements on the well-foundedness of transition invariants. We show that this strengthened property is enough to show the termination of any program. We say that a relation $T \subseteq S \times S$ is *well-founded on the reachable states* if $T \cap R^+ \cap (\mathscr{R} \times \mathscr{R})$ is well-founded.

Lemma 1. *A program P is terminating if and only if there exists a formula T that is a transition invariant well-founded on the reachable states.*

Proof. We provide the proof in Appendix C.1.

Lemma 2. *Let T be a disjunctively well-founded transition invariant of P, then T is a transition invariant well-founded on the reachable states.*

Proof. We provide the proof in Appendix C.2.

4.2 Ranking Functions to Transition Invariants

One common approach to proving termination is to synthesize ranking functions [69]. Ranking functions map the progress of a loop to a well-ordered set. However, when the loop is complex, the ranking function can be complex as well. A more general approach is to construct *lexicographic* ranking functions [70]. The termination argument is then based on constructing a disjunctively well-founded transition invariant from these functions [69]. In Lemma 3, we formulate a similar construction for the well-foundedness of the reachable states of P, which we later use in the validation.

Definition 4 (Lexicographic Ranking Function). *Let $R_L \subseteq S \times S$ be a transition relation of a program loop L. The sequence of functions $\langle \rho_0,...,\rho_k \rangle$, where $\rho_0,...,\rho_k : S \to \mathbb{N}_0$, is called a lexicographic ranking function for R_L if*

$$\forall (s,s') \in \mathscr{R} \times \mathscr{R} : R_L(s,s') \Rightarrow \exists 0 \leq i \leq k : \rho_i(s) > \rho_i(s') \wedge \forall 0 \leq j < i : \rho_j(s) \geq \rho_j(s')$$

Lemma 3. *Let L be a program loop. Functions $\rho_0,...,\rho_k : S \to \mathbb{N}_0$ form a lexicographic ranking function $\langle \rho_0,...,\rho_k \rangle$ of R_L, if and only if formula*

$$T(s,s') \equiv \bigvee_{0 \leq i \leq k} (\rho_i(s) > \rho_i(s') \bigwedge_{0 \leq j < i} \rho_j(s) \geq \rho_j(s'))$$

is a (inductive) transition invariant well-founded on the reachable states of the loop L.

Proof. We provide the proof in Appendix C.3.

Ranking functions are arguments that witness the well-foundedness of each loop in separation. However, transition invariants overapproximate the transition relation of the whole program. In the definition of semantics for termination witnesses in Sect. 3, we assign transition invariants to line numbers in the program. A transition invariant T in the witness is then valid if it holds for any two states s,s' that are reachable in the same program execution at the mapped line number. In the case of nested loops, exporting transition invariants from ranking functions in the form of Lemma 3 may not be correct with respect to the witness semantics. The problem is that the loop head of an inner loop for which a ranking function is computed is reachable not only through its own iterations but also through iterations of the outer loop. For example, consider the program in Fig. 2. CPAchecker computes the ranking function $\langle x \rangle$ for the inner loop and $\langle x-y \rangle$ for the outer loop. Following the construction in Lemma 3, the formula $x-y > x'-y'$ is a valid transition invariant for the inner loop at line 7, but it is not valid for the

whole program according to our semantics. This is because a state with $s(x)=3$ and $s(y)=2$ is reachable at line 7, and from there a state with $s'(x)=2$ and $s'(y)=0$ is also reachable, but $T(s,s')$ does not hold. To fix this issue, the ranking functions of the outer loops need to be included in a disjunction with the invariants for the inner loops. The correct transition invariant at line 7 is therefore $x-y>x'-y' \vee x>x'$.

4.3 Reachability Arguments to Transition Invariants

Another common approach to verify termination is to reduce it to a reachability problem [22, 35, 46–50]. We use the setting from Sect. 2.2 to show how to construct transition invariants from safety invariants obtained by a reachability verifier for the instrumented program.

Consider a program P with n variables, its instrumented program P', and a formula T with two sets of n free variables. We can view this formula in two ways. It induces a set of states in P', assigning values of variables from $\widehat{X}$ to the first and from X to the second set of variables, respectively. Alternatively, it represents a set of paths in P, assigning values of the first state to the first set of variables and values of the last state to the second set of variables. The following theorem formalizes that safe invariance and well-founded transition invariance are preserved when changing between these two views.

Theorem 3. *Let $P=(X,S,Init,R)$ be a program with finite state space S and $P'=(X',S',Init',R')$ be the instrumented program. A predicate*

$$I(\widehat{s}) \equiv T(\widehat{s}(\widehat{x}_0),...,\widehat{s}(\widehat{x}_n),\widehat{s}(x_0),...,\widehat{s}(x_n)) \text{ with } \widehat{s} \in S'$$

is a safe 1-step (inductive) invariant of P' if and only if

$$T(s,s') \equiv T(s(x_0),...,s(x_n),s'(x_0),...,s'(x_n)) \text{ with } s,s' \in S$$

is a (inductive) transition invariant of P that is well-founded on the reachable states.

Proof. We provide the proof of the base theorem in Appendix C.1 and the extension for inductivity in Appendix C.6.

Lemma 4. *Let $P'=(X',S',Init',R')$ be the instrumented program for program P. Formula $I(\widehat{x}_0,...,\widehat{x}_n,\widehat{pc},x_0,...,x_n,pc,saved)$ is a safe 1-step invariant of P' if and only if $I^t(\widehat{s})$ given as $\bigvee_{l,l' \in D_{pc}} I(\widehat{s}(\widehat{x}_0),...,\widehat{s}(\widehat{x}_n),l,\widehat{s}(x_0),...,\widehat{s}(x_n),l',1)$ is too.*

Proof. We provide the proof in Appendix C.4.

Putting together Theorem 3 and Lemma 4, we can construct a transition invariant well-founded on the reachable states of P from a safe invariant I, as follows:

$$T(s,s') \equiv \bigvee_{l,l' \in D_{pc}} I(s(\widehat{x}_0),...,s(\widehat{x}_n),l,s'(x_0),...,s'(x_n),l',1)$$

In the witness format [11], the safe invariants are related to specific locations, so, we substitute pc and $\widehat{pc}$ with the respective location. On the other locations, we trivially assume that the invariant is *false*.

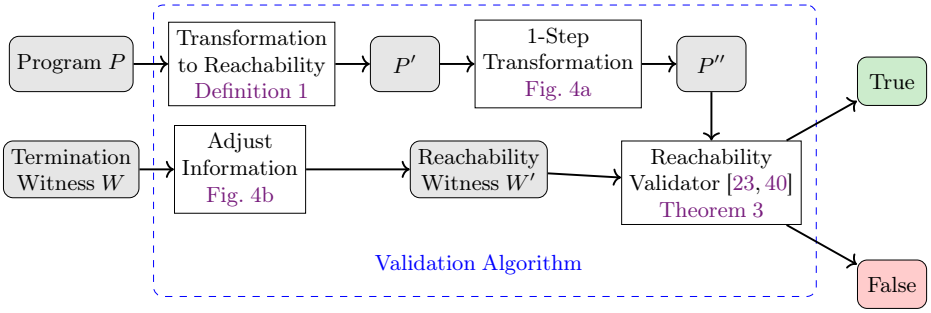


Fig. 3: Validation workflow via reachability transformation.

5 Validation Approaches

We present three validation approaches in the following section. The approach in Sect. 5.1 reduces the validation of termination witnesses to the validation of reachability witnesses [11], using the result of Theorem 3, thereby leveraging the efficiency of existing validators. The reachability witnesses contain only safety invariants and the validation have extensive support in existing tools [40, 71, 72]. Our reduction method uses tool TransVer [50] to transform the program, which has limited support for some C constructs such as arrays or structs.

Therefore, we present two other approaches that are not based on program transformation. They both follow the same workflow. For each formula in the termination witness, they check (weak) transition invariance and (disjunctive) well-foundedness. The two approaches differ in the well-foundedness checks. The method in Sect. 5.2.1 encodes the well-foundedness check into quantified SMT formulas. Such checks are usually quite efficient, but they are restricted to formulas with variables with finite domains. Hence, we introduce the second well-foundedness check in Sect. 5.2.2 that supports the validation of formulas with infinite domains.

5.1 Validating Termination Witnesses via Program Transformation

Reachability witnesses in the software verification format [11] contain set of safety invariants. A reachability witness is *valid* if all the invariants hold for all the reachable states at the given program location and conditions marked by `assert` hold for all the program executions.

Theorem 3 provides a direct way to reduce the validation of termination witnesses to the validation of reachability witnesses. Given a program with finite state space and a termination witness with transition invariants, we can transform the original program using the reduction from Definition 1 and validate that the transition invariants are safe 1-step invariants of the transformed program. Thanks to Lemma 2, we can validate also disjunctively well-founded transition invariants this way.

Figure 3 shows how to validate a termination witness using a reachability validator. The reachability validator must validate that the invariants are 1-step

```
1  int main(void) {
2      int saved = 0; int pc = 0;
3      int i1, j1;
4      int i, j;
5      j = 1;
6      i = 10000;
7      int first_1 = 0;
8      while (i-j >= 1) {
9        first_1 = 1;
10       if(saved == 0) {pc = 0; i1 = i; j1 = j;}
11       if(nondet() && saved == 0) {
12         saved = 1; pc = 1; i1 = i; j1 = j;
13       } else{
14         assert(saved == 0 || pc != 0
15           || i1 != i || j1 != j);
16       }
17       j = j + 1;
18       i = i - 1;
19     }
20     return 0;
21 }
```

(a) 1-Step Instrumentation

```
- entry_type: invariant_set
  metadata: ...
  content:
  - invariant:
      type: "loop_invariant"
      location:
        line: 8
      value: |
        "first_1 == 0 ||
        (i - j < i1 - j1)"
      format: "c_expression"
```

(b) Transformed Reachability Witness

Fig. 4: Program from Fig. 1b transformed with 1-step transformation (left), reachability witness encoding transition invariant as loop invariant (right)

invariants for the transformed program. Therefore, we further instrument the program by adding a variable `first_i` for each loop, where i is a loop-unique index. The variable is set to 1 after at least one iteration of the loop. We call this the *1-step transformation*, and Fig. 4a shows an example of its application to the transformed program from Fig. 1b.

The validation algorithm then transforms the termination witness. Every transition invariant T for the loop with index i is replaced with the safety invariant $\texttt{first\_i} = 0 \vee T$. This guarantees that the invariant is checked only after at least one iteration of the loop. It also replaces variables marked with `\at(var, AnyPrev)` by *var1* to match the ghost variables in the transformed program. Lastly, the line numbers are adjusted to match the transformed program. The produced witness and the final transformed program are then passed to the reachability validator, whose result is returned as the result of the validation.

Example 2. Consider the program P from Fig. 1a and its instrumented version P' from Fig. 1b. For program P, formula $T \equiv i - j > i' - j'$ overapproximates paths starting and ending at line 5. We can validate that T forms a well-founded transition invariant of the loop at line 5 in our witness by showing that $I \equiv \texttt{first\_1} = 0 \vee i1 - j1 > i - j$ is a safe invariant at line 8 for program in Fig. 4a.

5.2 General Validation Framework

Algorithm 1 presents a general validation framework of the termination witnesses. The algorithm tries to first validate the witness using Theorem 2 on line 7 and then Theorem 1 in the checks between line 9 and line 13. We first describe how the algorithm determines whether a given formula T is a (weak) transition invariant, and then in Sect. 5.2.1 and Sect. 5.2.2 we discuss the two different well-foundedness

checks. In all of the checks, we use the possible supporting safety invariants to overapproximate reachable states $\mathscr{R} \times \mathscr{R}$ with $I_L \equiv I_{L,0} \wedge ... \wedge I_{L,k}$.

(Weak) Transition Invariant. To validate that the witness certifies program termination according to Theorem 2, we need to check that the formula T is a weak transition invariant. This can be done using an SMT solver to check that the formula is implied after one iteration of the loop:

$$\texttt{is\_weak\_TI}(R_L,T,I_L) \equiv \forall s,s' \in S : R_L(s,s') \wedge I_L(s) \wedge I_L(s') \Rightarrow T(s,s')$$

To validate that formula T is a transition invariant, we propose an algorithm that checks inductivity of T. This is sufficient as an inductive relation with respect to R_L is a transition invariant [13]. However, not every transition invariant is an inductive relation [13]. Therefore, we propose a more general check following a similar idea as k-induction [73] for safety properties. The function $\texttt{is\_TI}(R_L,T,I_L)$ consists of an infinite loop that after each iteration increments a variable k that is initially set to $k = 1$. In each iteration, it does the following checks:

$$\textbf{Base Case: } \forall 1 \leq i \leq k : \forall s_0,...,s_i : \Big(\bigwedge_{0 \leq j \leq i} I_L(s_j) \bigwedge_{0 \leq j < i} R(s_j,s_{j+1}) \Big) \Rightarrow T(s_0,s_i)$$

$$\textbf{Step Case: } \forall s,s_0,...,s_k : \Big(I_L(s) \wedge T(s,s_0) \bigwedge_{0 \leq j \leq k} I_L(s_j) \bigwedge_{0 \leq j < k} R(s_j,s_{j+1}) \Big) \Rightarrow T(s,s_k)$$

If any of the formulas from the base case is not valid, we can confirm that T is not a transition invariant. If the formula for the step case is not valid, we increment k and continue with the next set of formulas.

Lemma 5. *If the base-case and step-case formulas are valid for some* $1 \leq k$ *as checked by* `is_TI(R_L, T, I_L)`*, then* T *is a transition invariant of the loop* L*.*

Proof. We provide the proof in Appendix D.1.

(Disjunctively) Well-Founded. We propose two methods for checking well-foundedness. If the program contains only variables over finite domains, we can leverage the efficiency of SMT solvers and reduce the check to SMT queries with quantifiers. The second method is more general and can also handle programs with variables over infinite domains. It checks the well-foundedness by encoding formula T into a new program which is terminating if T is well-founded. We can then use verifiers that can handle infinite domains to validate the well-foundedness of T. To check disjunctive well-foundedness, we first put the formula into *disjunctive normal form (DNF)* and then check that each clause is well-founded.

5.2.1 Well-Foundedness Check: Decreasing Set of Reachable States.

The first approach for well-foundedness checks is called *decreasing set*. It reduces checking the well-foundedness of T to checking the validity of the following SMT formula with quantifiers:

$$\texttt{is\_well\_founded}(T,I_L) \equiv T(s,s') \wedge I_L(s) \wedge I_L(s') \Rightarrow$$
$$\Big(\big(\exists s_0 \in S : T(s,s_0) \wedge \neg T(s',s_0) \wedge I_L(s_0) \big) \ \wedge \ \big(\forall s_0 \in S : T(s',s_0) \wedge I_L(s_0) \Rightarrow T(s,s_0) \big) \Big)$$

Algorithm 1 General validation framework

Input: a program P, a termination witness W
Output: `true` if the witness is correct, `false` if the witness incorrect, and `unknown` if the validator could not decide

```
1: for L∈LOOPS(P) do
2:    (L,T,{I_{L,0}...I_{L,k}}) ← extract_for_loop(W,L);
3:    for 1≤i≤k do
4:       if ¬is_supporting_invariant(I_{L,i},R_L) then
5:          return false;
6:    I_L ← I_{L,0}∧...∧I_{L,k};
7:    if is_well_founded(T,I_L)∧is_weak_TI(R_L,T,I_L) then
8:       continue;
9:    T ← DNF(T);
10:   for C∈CLAUSES(T) do
11:      if ¬is_disj_well_founded(C,I_L) then
12:         return unknown;
13:   if ¬is_TI(R_L,T,I_L) then
14:      return false;
15: return true;
```

It follows the idea of implicit ranking functions [74]. If the formula is valid, it proves that the size of the set of successors $succ(s) = \{s' \in S \mid T(s,s') \wedge I_L(s')\}$ decreases. In other words, it holds for reachable states that $T(s,s') \Rightarrow |succ(s)| > |succ(s')|$. An additional completeness requirement is that T contains only variables with finite domains, even though the program itself may include variables with infinite domains. It is needed, so that $|succ(s')|$ eventually reaches zero. We show the correctness of the approach in the following lemma.

Lemma 6. *Let $T \subseteq S \times S$ be a relation over states from P expressed as a formula containing only variables with finite domains. The formula T is well-founded on the reachable states, if the formula* `is_well_founded`(T,I_L) *is valid.*

Proof. We provide the proof in Appendix D.2.

If the formula `is_well_founded`(T) is not valid, Alg. 1 attempts to decompose T into a disjunction $C_1 \vee ... \vee C_n$ and prove that each clause C in the disjunction is well-founded. However, since it first proves that T is a transition invariant, it is sufficient to use simpler queries:

$$\texttt{is\_disj\_well\_founded}(C,I_L) \equiv \forall s \in S : \neg(C(s,s) \wedge I_L(s))$$

Lemma 7. *Let $T \subseteq S \times S$ be a relation over states from P expressed with formula containing only variables with finite domains and with $R_L^+ \Rightarrow T$. Relation $R_L^+ \cap (\mathscr{R} \times \mathscr{R})$ is well-founded, if there exist formulas $C_1, ..., C_n$ such that $T = C_1 \vee ... \vee C_n$ and $\forall 1 \leq i \leq n$:* `is_disj_well_founded`(C_i,I_L).

Proof. We provide the proof in Appendix D.3.

Notice that $R_L^+ \cap (\mathscr{R} \times \mathscr{R})$ is a trivial transition invariant and hence, Lemma 7 validates the termination of L.

```
1  int main() {
2    int x = nondet();
3    int y = nondet();
4    int x__PREV = nondet();
5
6    while (x__PREV > 0 && ((y <= 1 && x <= 0) || ( x < x__PREV))) {
7      x__PREV = x;
8      x = nondet();
9      y = nondet();
10   }
11 }
```

Fig. 5: An example of program encoding a transition invariant

5.2.2 Well-Foundedness Check: Validation via Verification. We propose another approach to well-foundedness checks to support formulas with variables with infinite domains. The main idea is to reduce the problem of checking well-foundedness to proving termination of a simplified program. We can interpret a transition invariant T as the transition relation of a program. Such a program consists of a single loop that performs computations equivalent to the transition relation T. Proving the termination of this program is therefore equivalent to proving that T is well-founded. In practice, this is typically much simpler than proving termination of the original program, since T often abstracts away the complex logic of the program's loops. Most importantly, this allows us to apply any termination algorithm – including those that support infinite state spaces, such as ranking-function synthesis – to prove the termination of the simplified program, as shown in Lemma 8.

Lemma 8. *Let L be a program loop, and $T \subseteq S \times S$ be a relation over the states from program P. If the program with the transition relation $R_T(s, s') \equiv LC(s) \wedge I_L(s) \wedge I_L(s') \wedge T(s,s')$, where LC is the loop condition of L, is terminating, then T is well-founded.*

Proof. We provide the proof in Appendix D.4.

Both functions `is_well-founded`(T,I_L) and `is_disj-well-founded`(T,I_L) perform the same check. They encode R_T into a program with a single loop that performs a computation equivalent to the transition formula $LC(s) \wedge I_L(s) \wedge I_L(s') \wedge T(s,s')$, where LC is the loop condition of the original loop from P. An example of such a program is provided in Fig. 5. Afterwards, they apply an off-the-shelf termination verification algorithm to prove that $R_T(s,s')$ is well-founded.

Example 3. Consider the program from Fig. 2 and the invariant $(y' \leq 1 \wedge x' \leq 0) \vee (x' < x)$ from the witness for the loop at line 5. Figure 5 shows the program that is used by `is_well-founded`(T) to check the well-foundedness of the invariant.

6 Experimental Evaluation

Termination witnesses should aid in exchanging information between tools about the verification results. To analyze this, we first assess whether the information

transfer is effective, i.e., whether the produced witnesses can be validated by independent validators. We evaluate this in Item RQ 1 by measuring how many witnesses produced by the different construction approaches (Sect. 4) can be validated by the different validation approaches (Sect. 5). Second, we assess whether the information contained in the witnesses is useful for validators to prove termination, compared to verifiers that do not have this information. We evaluate this in Item RQ 2 by comparing verification and validation times, and the number of programs that a validator can solve thanks to the witnesses that the verifier, based on the same tool, cannot solve alone. We summarize these two assessment areas in the following research questions:

RQ 1 (Effectiveness): Can the produced witnesses be effectively validated by using the proposed validation approaches?

RQ 2 (Information Exchange): Do the produced witnesses aid validators in proving termination?

Benchmarks. To answer the research questions, we use the programs from SV-Benchmarks which is the largest publicly-available collection of C programs with expected verdicts for different properties. Our benchmark set consists of all the programs which are expected to terminate, i.e., have an expected verdict `true` for the property termination at the tag SV-COMP 2026.

Benchmark Environment. To ensure reliable measurements of the resource consumption, we use BenchExec [75]. We parallelize all the runs on a cluster consisting of machines with Intel Xeon E5-1230 CPUs and 32 GB of memory using BenchCloud [76]. Every run gets one physical core (two processing units), and 900 s of CPU time with 15 GB of memory for verification and validation.

Verification Tools. We implemented the conversion of ranking functions to transition invariants (Sect. 4.2) in CPAchecker [22] in version [Anonymized]. Further verifiers in our comparison use TransVer [50] to reduce termination to reachability (Definition 1), and then we use UAutomizer [40] and Goblint [30] in their SV-COMP 2026 version, and CPAchecker in its version [Anonymized] to verify the transformed programs. We transform the safety invariants contained in the witnesses produced for the transformed programs into transition invariants (Sect. 4.3) using TransVer in version [Anonymized].

Validation Tools. To validate the produced witnesses, we use MetaVal in version [Anonymized], to transform them to reachability witnesses (Sect. 5.1) using TransVer in version [Anonymized] for the transformation, and CPAchecker [77] or UAutomizer [40] as the reachability validator backend run through FM-Weck [78] in version [Anonymized]. We call them CPAchecker-TransVer × and UAutomizer-TransVer △. The other validation approaches, i.e., the ones from Sects. 5.2.1 and 5.2.2, are called CPAchecker-decreasing □, and CPAchecker-simplifying ◇, respectively, and we implemented them in CPAchecker [77] in version [Anonymized].

Table 2: Amount of witnesses validated by at least one validator generated by the given verifier, and the amount of witnesses validated by the respective validator; in grey, percent of validated witnesses and in green, amount of tasks validated by the validator but not by its corresponding verifier

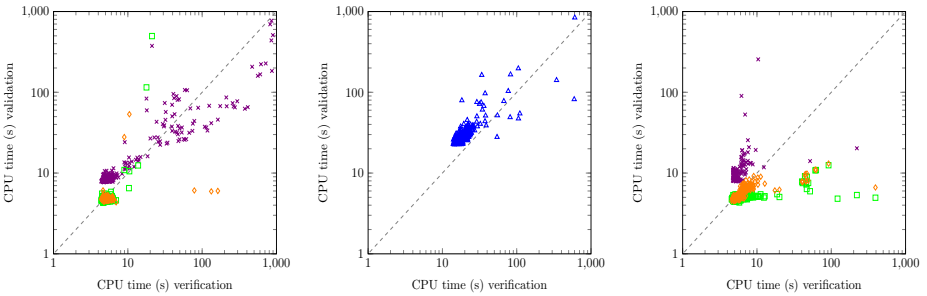
Verifier	Witnesses	Validated	Validator	Validated		Unique
CPAchecker TransVer	293	285 (97 %)	CPAchecker-TransVer ×	**281**	(0)	57
			UAutomizer-TransVer △	226	(5)↑	4
			CPAchecker-decreasing □	184	(94)↑	0
			CPAchecker-simplifying ◇	183	(96)↑	0
UAutomizer TransVer	482	473 (98 %)	CPAchecker-TransVer ×	354	(107)↑	3
			UAutomizer-TransVer △	**444**	(0)	66
			CPAchecker-decreasing □	362	(167)↑	4
			CPAchecker-simplifying ◇	275	(122)↑	1
Goblint TransVer	382	283 (74 %)	CPAchecker-TransVer ×	**250**	(4)↑	50
			UAutomizer-TransVer △	221	(5)↑	21
			CPAchecker-decreasing □	198	(99)↑	12
			CPAchecker-simplifying ◇	183	(97)↑	0
CPAchecker	422	406 (96 %)	CPAchecker-TransVer ×	131	(32)↑	0
			UAutomizer-TransVer △	253	(0)	48
			CPAchecker-decreasing □	**302**	(0)	1
			CPAchecker-simplifying ◇	286	(0)	0

6.1 RQ 1: Effectiveness

To analyze the effectiveness of the validators, we measure how many of the generated witnesses could be validated. Verifiers produce a witness only if they manage to successfully verify the program. The left side of Table 2 shows how many programs could be verified by each verifier, and hence, how many termination witnesses they produced. It also shows how many of these witnesses could be validated by at least one of the validators. The right side shows the numbers for the validation. The green numbers with arrows indicate how many more tasks the validator can solve compared to the verifier implemented in the same tool, using the witnesses from the verifier in the corresponding row. For example, $(94)^{\uparrow}$ in the first row indicates that CPAchecker-decreasing □ could prove 94 more programs to be terminating than CPAchecker, using witnesses from CPAchecker-TransVer.

Interestingly, CPAchecker-TransVer could only verify 293 programs, but CPAchecker-TransVer × could validate 354 witnesses of UAutomizer-TransVer. CPAchecker uses a subset of analyses for validation than it uses for verification. Hence, this difference in performance is attributed to the additional information in the witnesses, which will be further analyzed in RQ 2.

Altogether, 99 witnesses produced by Goblint-TransVer were not validated by any of the validators. The reason is that Goblint produced 4551 invariants in total, which is 12 invariants on average per program even though, they mostly



(a) CPAchecker-TransVer **(b)** UAutomizer-TransVer **(c)** CPAchecker

Validators	CPAchecker-TransVer (a)	UAutomizer-TransVer (b)	CPAchecker (c)
CPAchecker-TransVer ×	1.0	–	1.0
UAutomizer-TransVer △	–	1.0	–
CPAchecker-decreasing □	1.5×10^{-18}	–	7.8×10^{-49}
CPAchecker-simplifying ◇	8.2×10^{-13}	–	9.8×10^{-29}

(d) p-values

Fig. 6: Scatter plots **(a-c)** of the time taken for the validation of the termination witnesses (y-axis, seconds) vs. the time taken for the verification of the original/-transformed program (x-axis, seconds); Table **(d)** with p-values for a Wilcoxon signed-rank test for the hypothesis that CPU time of verification is equal to validation; numbers below 0.01 are displayed in green and above in red

contain at most 2 loops. Since the validators need to check each of them, most of them runs into timeout or out of memory.

Yes, the validation is effective, as a large portion of the produced witnesses can be validated by at least one validator.

6.2 RQ 2: Information Exchange

Transition invariants can abstract away complex loop behavior. Hence, given a useful transition invariant, it should be simpler to show that it is well-founded than to show that the loop in the program terminates.

The green numbers with arrows in Table 2 show that all validators benefit from the information in the witnesses. Each one could solve some tasks using the information from the witnesses that their corresponding verifier could not solve alone. Most notably, CPAchecker-decreasing □ and CPAchecker-simplifying ◇, using the witnesses from the transformed programs, could solve around 100 tasks which CPAchecker in its verification mode could not solve. The information transfer is also not tool-specific as these validators could solve more tasks using for example, the witnesses from Goblint-TransVer. The reason for this large improvement is that the different tools solve different sets of tasks. There-

fore, providing the information from one tool to another, helps it in solving tasks that it could not solve alone.

To further evaluate this, we compare the CPU time of verification and validation approaches using the same backend tools to mitigate the influence of implementation details such as JVM start-up time or framework-specific inefficiencies.

Figure 6 presents the scatter plots comparing the CPU time for verifier and validator approaches using CPAchecker and UAutomizer backends on commonly solved tasks. Since it is often difficult to infer the density of point clusters in scatter plots, we additionally report p-values for the *Wilcoxon signed-rank test* [79] in Fig. 6d. We pose the null hypothesis that the verification time is less or equal to the validation time. Our alternative hypothesis states that the verification time is stochastically greater than the validation time. The lower the p-value, the smaller the probability of falsely rejecting the null hypothesis and, hence, the higher the probability of correctly confirming the alternative hypothesis.

The results of the statistical test show that the general validation framework implemented in CPAchecker, with both well-foundedness checks (CPAchecker-decreasing □ and CPAchecker-simplifying ◇), is on average faster at validating the transition invariants obtained from ranking functions and safety invariants than CPAchecker's verification algorithms. Validation through reduction to reachability witnesses is less efficient for both UAutomizer and CPAchecker. An interesting observation from the scatter plots is that if we restrict the CPAchecker data to difficult verification tasks, i.e., those that took at least 10s to verify, all three validation techniques are more efficient than verification.

Yes, tools can solve more tasks using the information contained in the witnesses. Furthermore, the validation methods from Sects. 5.2.1 and 5.2.2, implemented in CPAchecker, are faster on average than their verification algorithms. However, the validation methods based on transformation from Sect. 5.1 are only more efficient for tasks where the verification took more than 10 seconds.

7 Conclusion

We proposed to use *transition invariants* as a unifying formalism for termination witnesses that is both *general* and efficient to *validate*. We demonstrated how to systematically construct transition invariants from common termination arguments and extended the software witness exchange format 2.0 to support them. Furthermore, we introduced three complementary validation approaches: (a) reducing the validation of termination witnesses to the validation of reachability witnesses, (b) encoding validation as quantified SMT queries and exploiting the implicit ranking of a decreasing reachable set, and (c) verifying the termination of a program with a single loop that encodes the transition invariant. We formally proved the correctness of all construction and validation methods. Our experimental results show that the majority of termination witnesses can be validated effectively and, in many cases, validation is more efficient than verification. So far, no exchange format for termination witnesses for software verification existed,

and our proposal was already adopted by the SV-COMP community. We hope that many researchers and practitioners find our format and approaches useful and can now support and use termination witnesses.

Data-Availability Statement. A reproduction package, which includes all software and data that we used for our experiments, is available on Zenodo [80].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY).

Acknowledgements. We thank the verification community for supporting our witness format. The format was incorporated into the common witness format of the software-verification community, version 2.1 (`https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/`). The competition on software verification supported our termination witnesses in SV-COMP 2026 [81] as demo track, and will officially support termination witnesses in SV-COMP 2027. Since witnesses are mandatory in SV-COMP, there will be many verification tools supporting it.

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT (1999)
2. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
3. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
4. Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design. pp. 554–582. LNCS 13660, Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27
5. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with Slam. Commun. ACM **54**(7), 68–76 (2011). https://doi.org/10.1145/1965724.1965743
6. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Proc. CAV. pp. 415–418. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_37
7. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. PLDI. pp. 415–426. ACM (2006). https://doi.org/10.1145/1133981.1134029
8. Giacobbe, M., Kröning, D., Parsert, J.: Neural termination analysis. In: Proc. ESEC/FSE. pp. 633–645. ACM (2022). https://doi.org/10.1145/3540250.3549120
9. 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
10. 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
11. 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). https://doi.org/10.1007/978-3-031-66149-5_11
12. 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
13. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proc. LICS. pp. 32–41. IEEE (2004). https://doi.org/10.1109/LICS.2004.1319598

14. Beyer, D., Strejček, J.: SV-Witnesses — Format version 2.1. Zenodo (2025). https://doi.org/10.5281/zenodo.17277275
15. 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). https://doi.org/10.1007/978-3-031-30820-8_31
16. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9
17. Lommen, N., Giesl, J.: AProVE (KoAT+LoAT) (competition contribution). In: Proc. TACAS (3). pp. 205–211. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_13
18. Chalupa, M., Richter, C.: Bubaak: Dynamic cooperative verification (competition contribution). In: Proc. TACAS (3). pp. 212–216. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_14
19. Chalupa, M., Henzinger, T.: Bubaak: Runtime monitoring of program verifiers (competition contribution). In: Proc. TACAS (2). pp. 535–540. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
20. Chalupa, M., Richter, C.: Bubaak-SpLit: Split what you cannot verify (competition contribution). In: Proc. TACAS (3). pp. 353–358. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_20
21. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
22. 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 (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
23. 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). https://doi.org/10.1007/978-3-031-71177-0_30
24. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
25. 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). https://doi.org/10.1007/978-3-031-90660-2_15
26. Bajczi, L., Szekeres, D., Mondok, M., Ádám, Zs., Somorjai, M., Telbisz, C., Dobos-Kovács, M., Molnár, V.: EmergenTheta: Verification beyond abstraction refinement (competition contribution). In: Proc. TACAS (3). pp. 371–375. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_23
27. Wu, T., Li, X., Manino, E., Menezes, R., Gadelha, M., Xiong, S., Tihanyi, N., Petoumenos, P., Cordeiro, L.: ESBMC v7.7: Efficient concurrent software verification with scheduling, incremental SMT and partial order reduction (competition contribution). In: Proc. TACAS (3). pp. 223–228. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_16
28. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via *k*-induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (February 2017). https://doi.org/10.1007/s10009-015-0407-9

29. Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination (competition contribution). In: Proc. TACAS (3). pp. 381–386. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_25
30. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
31. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-C with trace partitioning and autosuggestions (competition contribution). In: Proc. TACAS (3). pp. 229–235. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_17
32. Lemberger, T., Wachowitz, H.: Nacpa: Native checking with parallel-portfolio analyses (competition contribution). In: Proc. TACAS (3). pp. 236–241. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_18
33. Chaudhary, E., Joshi, S.: Pinaka: Symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS (3). pp. 234–238. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_20
34. 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). https://doi.org/10.1007/978-3-031-90660-2_19
35. 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). https://doi.org/10.1007/978-3-031-57256-2_27
36. 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). https://doi.org/10.1007/978-3-031-57256-2_29
37. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
38. 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). https://doi.org/10.1007/978-3-031-90660-2_22
39. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: A framework for abstraction refinement-based model checking. In: Proc. FMCAD. pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257
40. 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). https://doi.org/10.1007/978-3-031-57256-2_31
41. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
42. 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). https://doi.org/10.1007/978-3-031-90660-2_9
43. Cook, B., Kröning, D., Rümmer, P., Wintersteiger, C.: Ranking function synthesis for bit-vector relations. In: Proc. TACAS. pp. 89–103. Springer (2010)
44. Leike, J., Heizmann, M.: Ranking templates for linear loops. Logical Methods in Computer Science **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:16)2015

45. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Proc. ATVA. pp. 365–380. LNCS 8172, Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_26
46. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electr. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006). https://doi.org/10.1016/j.entcs.2005.11.018
47. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: Proc. FMICS. pp. 160–177. No. 2 in ENTSC 66, Elsevier (2002). https://doi.org/10.1016/S1571-0661(04)80410-9
48. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
49. Chen, J., He, F.: Proving termination by k-induction. In: Proc. ASE. pp. 1239–1243. ACM (2021). https://doi.org/10.1145/3324884.3418929
50. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Xia, T., Zheng, X.: TransVer: A modular program-transformation framework for reduction to reachability. In: Proc. SPIN. pp. 1–24. LNCS 15945, Springer (2025). https://doi.org/10.1007/978-3-032-06847-7_1
51. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: Proc. POPL. pp. 132–144. ACM (2005). https://doi.org/10.1145/1040305.1040317
52. Urban, C., Gurfinkel, A., Kahsai, T.: Synthesizing ranking functions from bits and pieces. In: Proc. TACAS. pp. 54–70. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_4
53. Yao, J., Tao, R., Gu, R., Nieh, J.: Mostly automated verification of liveness properties for distributed protocols with ranking functions. POPL **8** (2024). https://doi.org/10.1145/3632877
54. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proc. LPAR. pp. 348–370. LNCS 6355, Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
55. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: FM. LNCS, vol. 8442, pp. 127–131. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_9
56. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. NFM. pp. 41–55. LNCS 6617, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
57. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
58. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The key tool. Software and System Modeling **4**(1), 32–54 (2005). https://doi.org/10.1007/s10270-004-0058-x
59. Brummayer, R., Biere, A., Lonsing, F.: Btor: Bit-precise modelling of word-level problems for model checking. In: Proc. SMT/BPR. pp. 33–38. ACM (2008). https://doi.org/10.1145/1512464.1512472
60. Heule, M.J.H.: The DRAT format and DRAT-trim checker. CoRR **1610**(06229) (October 2016). https://doi.org/10.48550/arXiv.1610.06229
61. Cimatti, A., Griggio, A., Sebastiani, R.: A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In: Proc. SAT. pp. 334–339. LNCS 4501, Springer (2007). https://doi.org/10.1007/978-3-540-72788-0_32
62. 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

63. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2024. In: Proc. FMCAD. pp. 7–7. TU Wien Academic Press (2024). https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_6
64. Froleyks, N., Yu, E., Preiner, M., Biere, A., Heljanko, K.: Introducing Certificates to the Hardware Model Checking Competition. In: Proc. CAV. LNCS, vol. 15931, pp. 281–295. Springer (2025). https://doi.org/10.1007/978-3-031-98668-0_14
65. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT competition 2020. Artif. Intell. **301**, 103572:1–103572:25 (2021). https://doi.org/10.1016/j.artint.2021.103572
66. 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
67. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015-2018. J. Satisf. Boolean Model. Comput. **11**(1), 221–259 (2019). https://doi.org/10.3233/SAT190123
68. 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
69. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Proc. VMCAI. pp. 239–251. LNCS 2937, Springer (2004). https://doi.org/10.1007/978-3-540-24622-0_20
70. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 47–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_4
71. 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 (extended version). arXiv/CoRR (2024)
72. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
73. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
74. Lotan, R., Shoham, S.: Implicit rankings for verifying liveness properties in first-order logic. In: Proc. TACAS. pp. 375–395. Springer (2025). https://doi.org/10.1007/978-3-031-90643-5_20
75. 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
76. Beyer, D., Chien, P.C., Jankola, M.: BenchCloud: A platform for scalable performance benchmarking. In: Proc. ASE. pp. 2386–2389. ACM (2024). https://doi.org/10.1145/3691620.3695358
77. Beyer, D., Lingsch-Rosenfeld, M.: CPAchecker 4.0 as witness validator (competition contribution). In: Proc. TACAS (3). pp. 192–198. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_11
78. Beyer, D., Wachowitz, H.: FM-Weck: Containerized execution of formal-methods tools. In: Proc. FM. pp. 39–47. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_3
79. Woolson, R.: Wilcoxon Signed-Rank Test, pp. 1–3. John Wiley & Sons, Ltd (2008). https://doi.org/10.1002/9780471462422.eoct979

80. Anonymous: Reproduction package for CAV 2026 submission 'Transition invariants revisited: Termination witnesses and their validation'. Zenodo (2026). https://doi.org/10.5281/zenodo.18384699
81. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. TACAS (2). pp. 461–501. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_23
82. 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). https://doi.org/10.1007/978-3-031-57256-2_15
83. 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
84. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition power abstractions for deep counterexample detection. In: Proc. TACAS. pp. 524–542. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_29

Table 3: SV-COMP properties and their support by witness formats

Property	Correctness		Violation	
	version 1.0	version 2.0	version 1.0	version 2.0
Reachability	✓	✓	✓	✓
Memory Safety			✓	
Concurrency Safety			✓	
No Data Races			✓	
No Overflows	✓	✓	✓	✓
Termination			✓	

A Software Witnesses for Specifications

The rules of SV-COMP require participating verifiers to output a certificate called a witness to support the resulting verdict. If the verifier finds a bug, it outputs a violation witness. In the case that the program is correct, the verifier should output a correctness witness. There are currently two versions of the software witness format. Table 3 lists which competition specifications have a witness format in the version 2.0. We can see a clear gap in support of certifying termination proofs.

The verdict of a verifier is validated by a stand-alone tool called validator. Termination also lacks stand-alone validators that could validate termination proofs [82].

B Witness Format

Software verification witnesses [11, 83] are a well-established concept in the software verification community to transfer information between different tools. One of the most important uses is providing a machine-readable explanation for the verification result in the form of invariants for correctness arguments or traces for violations. Although witnesses in version 1.0 supported non-termination arguments [83], until now, no witness format has been expressive enough to encode termination arguments. In order to do this, we extend the existing verification witnesses in version 2.0 [11] to support transition invariants[2]. Since transition invariants can be a powerful argument for more than only termination arguments [13, 51, 84], we present the semantics of transition invariants for any specification.

We propose to add two new `invariant` entry type into the witness format version 2.0: `loop_transition_invariant` and `location_transition_invariant`. The new entry contains a value of type `c_expression` which represents the transition invariant. The value must be a side-effect free C expression over variables in scope at the indicated location. Moreover, the format allows `\at(x, AnyPrev)` for every variable `x` in the current scope. The value of `\at(x, AnyPrev)` is the value of `x` in some previous state of the program at that location. A transition invariant is valid if its value is true for every path through the program that visits

[2] The merge request proposing the changes: *Not included due to anonymity*

the location at least twice with `\at(var, AnyPrev)` referring to the value of the variables in some previous state from the path at that location. Note that `\at(var, AnyPrev)` is not well-defined the first time the location is visited, therefore transition invariants should only be evaluated when the location has been visited at least twice. Additionally, `\at(var, AnyPrev)` always refers to the same previous state of the program at the location for all variables i.e. `\at(x, AnyPrev)` and `\at(y, AnyPrev)` refer to the values of `x` and `y` in the same previous state of the program.

To summarize, for a witness with transition invariants to be valid the following conditions must be met:

- All invariants must hold for every path through the program
- The program must fulfill the specification

Example 4. Figure 2 shows an example program and a correctness witness with transition invariants corresponding to T_0 and T_1 from the witness, providing an argument that the program terminates. Termination of the inner loop L_1 is explained by T_1 using the fact that either the value of `y` is increasing in every iteration, expressed by `\at(y, AnyPrev) + 1 <= y`, or the value of `x` has decreased from the last visit of the loop, implied by `x + 1 <= \at(x, AnyPrev)`. Hence, no state is visited twice. The outer loop terminates because L_1 always terminates and the value of `x` is decreasing, expressed by `x < \at(x, AnyPrev)`.

C Proofs for Sect. 4

C.1 Proof of Lemma 1

Proof. Direction $\Leftarrow$: Let us prove this direction by contradiction. Let T be a transition invariant that is well-founded on the reachable states and let P be non-terminating. Hence, there exists an infinite sequence $\langle s_0,s_1,...\rangle$ of states, such that $\mathit{Init}(s_0) \wedge \forall i \geq 0 : R(s_i,s_{i+1})$ holds. Since $\mathit{Init}(s_0)$, all the states from the sequence are reachable $(s_0,s_1,... \in \mathscr{R})$. Moreover, T is a transition invariant and therefore, $\forall i \geq 0 : (s_i,s_{i+1}) \in (T \cap (\mathscr{R} \times \mathscr{R}) \cap R^+)$ which is a contradiction to T being well-founded on the reachable states.

Direction $\Rightarrow$: The same argument as Podelski and Rybalchenko used in their proof that Theorem 1 holds [13]. □

C.2 Proof of Lemma 2

Proof. If T is a disjunctively well-founded transition invariant of P, then P is terminating according to Theorem 1. Hence, the transitive closure of the transfer relation restricted to reachable states $R^+ \cap (\mathscr{R} \times \mathscr{R})$ of P is well-founded. Therefore, the relation $(T \cap (\mathscr{R} \times \mathscr{R}) \cap R^+)$ is also well-founded. □

C.3 Proof of Lemma 3

Proof. Let L be a program loop, $\rho_0,...,\rho_k : S \to \mathbb{N}_0$ be functions and formula T be

$$T(s,s') \equiv \bigvee_{0 \leq i \leq k} (\rho_i(s) > \rho_i(s') \bigwedge_{0 \leq j < i} \rho_j(s) \geq \rho_j(s'))$$

Direction $\Rightarrow$: *Transition invariant:* This directly follows from the Definition 4. *Inductivity (the theorem can be extended with inductivity):* Let us assume that $T(s,s') \wedge R_L(s',s'')$ holds for some $s,s',s'' \in S$. In the following, we show that $T(s,s'')$ holds, as well. Since $T(s,s')$, then there exists such $0 \leq i \leq k$ that $\rho_i(s) > \rho_i(s') \bigwedge_{0 \leq j < i} \rho_j(s) \geq \rho_j(s')$. From Definition 4 and $R_L(s',s'')$, we know that there exists i' such that $\rho_{i'}(s) > \rho_{i'}(s') \bigwedge_{0 \leq j < i'} \rho_j(s) \geq \rho_j(s')$. We further divide the proof by the following cases:

- $i \leq i'$, it holds that $\rho_i(s) > \rho_i(s')$ and $\rho_i(s') \geq \rho_i(s'')$. Hence, it holds that $\rho_i(s) > \rho_i(s'')$.
- $i > i'$, it holds that $\rho_{i'}(s) \geq \rho_{i'}(s')$ and $\rho_{i'}(s') > \rho_{i'}(s'')$. Hence, it holds that $\rho_{i'}(s) > \rho_{i'}(s'')$.

Well-foundedness on the reachable states: We prove the property by contradiction. Let $s_0,s_1,...$ be an infinite sequence with $Init(s_0) \wedge \forall i \geq 0 : T(s_i,s_{i+1}) \wedge R_L^+(s_i,s_{i+1})$. However, since $\rho_0,...,\rho_k$ map the states to $\mathbb{N}_0$ and for every transition $T(s_i,s_{i+1})$ at least one of them decreases, eventually, there is a state s_j in the sequence, which is mapped to 0 by every ranking function. Trivially, such a state does not have a successor in T.
Direction $\Leftarrow$: Let us prove this by contraposition. We assume that $\rho_0,...,\rho_k$ do not form a lexicographic ranking function for the loop L. Hence, there are states $(s,s') \in \mathscr{R} \times \mathscr{R}$ such that $R_L(s,s')$ and for all $0 \leq i \leq k$, it holds that $\rho_i(s) \leq \rho_i(s')$ or there is $0 \leq j < i$ such that $\rho_j(s) < \rho_j(s')$. Therefore, $T(s,s')$ also does not hold and therefore, T is not a transition invariant. □

C.4 Proof of Lemma 4

Proof. Direction $\Rightarrow$: *1-step invariant:* Assume two states $\widehat{s},\widehat{s}' \in \mathscr{R}'$, such that $R^+(\widehat{s},\widehat{s}')$. Since we assume any combination of locations in I^t, it is not relevant what the variables pc and $\widehat{pc}$ are assigned to in $\widehat{s}'$. We distinguish two cases:

- $\widehat{s}'(saved) = 1$, it follows that $I^t(\widehat{s}') \equiv I(\widehat{s}')$ and $I(\widehat{s}')$ is satisfied because I is 1-step invariant.
- $\widehat{s}'(saved) = 0$, in this case, $\widehat{s}_{\downarrow_{\widehat{X}}} = \widehat{s}'_{\downarrow_{\widehat{X}}}$ because no state has been saved in $\widehat{s}_{\downarrow_{\widehat{X}}}$. However, since both $\widehat{s}$ and $\widehat{s}'$ are reachable, there has to be a previous state $\widehat{s}''$ that can reach them both and $\widehat{s}_{\downarrow_{\widehat{X}}} = \widehat{s}''_{\downarrow_X}$. Hence, if we save this state, right after the first transition, states with the same mapping values as $\widehat{s}$ and $\widehat{s}'$, but with $\widehat{s}'(saved) = 1$ are still reachable from $\widehat{s}''$. Therefore, $I^t(\widehat{s}')$ also holds.

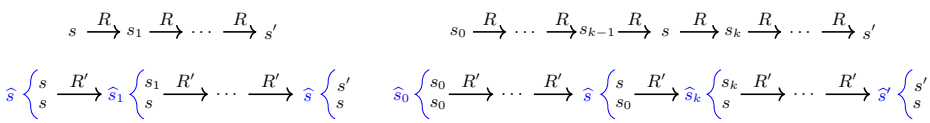


Fig. 7: A visual illustration of the first (left) and the second (right) case from the proof of transition invariance in Theorem 3. Every state from the instrumented program is divided into valuation of variables from X (up) and from $\widehat{X}$ (down).

Safety: Let us assume $I(\widehat{s})$ holds, we show that $\neg\widehat{s}(saved) \vee \bigvee_{x\in X} \widehat{s}(x) \neq \widehat{s}(\widehat{x})$ is satisfied. The case $\widehat{s}(saved) = 0$ is trivial. If $\widehat{s}(saved) = 1$, then $I^t(\widehat{s}) = I(\widehat{s})$ and from the safety of I, we get that $\varphi'(\widehat{s})$ holds.

Direction $\Leftarrow$: Let $I^t \equiv \bigvee_{l,l'\in D_{pc}} I(\widehat{s}(\widehat{x}_0),...,\widehat{s}(\widehat{x}_n),l,\widehat{s}(x_0),...,\widehat{s}(x_n),l',1)$ be an 1-step safe invariant of P'. Since I^t instantiates some of the variables of I but is the same otherwise, it holds that $I^t \Rightarrow I$. Therefore, I is an 1-step invariant, as well. Lastly, I is safe because I^t enumerates all the options for the *pc* variable in the disjunction and the only assignment it does not consider is $saved = 0$. However, in such case, the property is trivially satisfied, hence, I is safe. □

C.5 Proof of Theorem 3

Proof. In this proof, R and R' are transition formulas of P and P', respectively. Direction $\Rightarrow$: Let I be a safe 1-step invariant of instrumented program P'.
Transition invariant: We have to show the following

$$\forall s,s' \in (\mathscr{R}\times\mathscr{R}) : (R^+(s,s') \Rightarrow T(s,s'))$$

Let $\widehat{s}'$ be a state of instrumented program P' with $\widehat{s}'(\widehat{x}) = s(x)$ for all $\widehat{x}\in\widehat{X}$ and $\widehat{s}'(x) = s'(x)$ for all $x \in X$. In other words, it maps the ghost variables to the values assigned by s and the original variables to the values from s'. From the definition, if $I(\widehat{s}')$ holds, then we get $T(s,s')$. Hence, it is sufficient to show that $\widehat{s}'$ is reachable in at least one step in P'. Since, $s\in\mathscr{R}$, there are two cases possible (with a visual illustration in Fig. 7):

- $Init(s)$ holds. There is a state $\widehat{s}$, such that $\forall x \in X : \widehat{s}(x) = \widehat{s}(\widehat{x}) = s(x)$, $\widehat{s}(saved) = 0$ and hence $Init'(\widehat{s})$. More importantly, $\widehat{s}\in\mathscr{R}'$ and from the definition of R', $\widehat{s}$ can reach $\widehat{s}'$, i.e., $R'^+(\widehat{s},\widehat{s}')$. The path from $\widehat{s}$ to $\widehat{s}'$ follows the transitions from the path given by $R^+(s,s')$ and does not save a new state. Hence, it propagates s in the ghost variables from $\widehat{s}$ to $\widehat{s}'$. Lastly, from Definition 2, $I(\widehat{s}')$ holds.
- $\exists s_0,...,s_k : Init(s_0) \wedge R(s_0,s_1) \wedge ... \wedge R(s_{k-1},s) \wedge R(s,s_k)$ holds. Hence, there is a state $\widehat{s}_k$, such that $\forall x \in X : \widehat{s}_k(x) = s_k(x) \wedge \widehat{s}_k(\widehat{x}) = s(x)$, $\widehat{s}_k(saved) = 1$ and trivially, $\widehat{s}_k \in \mathscr{R}'$. Since $R(s,s_k)$, we can copy the state s into the ghost variables of $\widehat{s}_k$. Moreover, since $R^+(s,s')$, we can again pass s, further in the ghost variables and eventually reach $\widehat{s}'$. Therefore, $R'^+(\widehat{s},\widehat{s}')$ and $I(\widehat{s}')$ is satisfied.

Well-foundedness on the reachable states: We show the property by contradiction. Let $\langle s_0,s_1,...\rangle$ be an infinite sequence such that $Init(s_0) \wedge \forall i \geq 0 : T(s_i,s_{i+1}) \wedge$

$R^+(s_i,s_{i+1})$ holds. The domains of all the variables in P are finite. Hence, the set S of all the states is finite. Therefore, there exist at least two indices $j<k$, such that $s_j=s_k=s$. Since s is repeating, without loss of generality, we can assume that the line number $s(pc)$ is a loop-head of some loop of P. Moreover, since $R^+(s_j,s_k)$, we can save the state s_j thanks to the transtition $R(s_j,s_{j+1})$ in the state of P'. Therefore, state $\widehat{s}\in S'$ with $\forall x\in X:\widehat{s}(x)=\widehat{s}(\widehat{x})=s_j(x)$ and $\widehat{s}(saved)=1$ is reachable. However, this implies $I(\widehat{s})$, which contradicts that I is a safe invariant.

Direction $\Leftarrow$: Let $T(s,s')$ be a transition invariant that is well-founded on the reachable states of P.

1-step invariant: Assume two states $\widehat{s},\widehat{s}'\in\mathscr{R}'$, such that $R'^+(\widehat{s},\widehat{s}')$. Since $\widehat{s}$ is reachable, there exists a sequence $\langle\widehat{s}_0,...,\widehat{s}_j,\widehat{s}_{j+1},...,\widehat{s}_k\rangle$ of states, such that

$$Init'(\widehat{s}_0)\wedge R'(\widehat{s}_0,\widehat{s}_1)\wedge...\wedge R'(\widehat{s}_j,\widehat{s})\wedge R'(\widehat{s},\widehat{s}_{j+1})...\wedge R'(\widehat{s}_k,\widehat{s}')$$

From the construction of $Init'$ and R' in Definition 1, it follows that there is a state $\widehat{s}''$ in the sequence, such that $\widehat{s}'$ maps the ghost variables to the same values as $\widehat{s}''$ maps the original variables. More formally, $\widehat{s}''$ is either $\widehat{s}$ or is equal to some $\widehat{s}_i$ from the sequence, and $\forall x\in X:\widehat{s}''(x)=\widehat{s}'(\widehat{x})$. Further, $R^+(\widehat{s}''_{\downarrow X},\widehat{s}'_{\downarrow X})$ also follows from the definition of R'. Since T is a transition invariant of P, we get that $T(\widehat{s}''_{\downarrow X},\widehat{s}'_{\downarrow X})$ also holds. Therefore, $I(\widehat{s}''(\widehat{x}_0),...,\widehat{s}''(\widehat{x}_n),\widehat{s}'(x_0),...,\widehat{s}'(x_n))\equiv I(\widehat{s}'(\widehat{x}_0),...,\widehat{s}'(\widehat{x}_n),\widehat{s}'(x_0),...,\widehat{s}'(x_n))\equiv I(\widehat{s}')$ holds.

Safe invariant: Analogous to the proof of the *well-foundedness.* □

C.6 Proof of Theorem 3 - Inductivity

Proof. Direction $\Rightarrow$: The 1-step invariance of I follows from Lemma 1 and the base version of Theorem 3. Let T be an inductive transition invariant. Further, let $\widehat{s},\widehat{s}'\in S'$ be such that $R'(\widehat{s},\widehat{s}')\wedge I(\widehat{s})$. Transition $R'(\widehat{s},\widehat{s}')$ implies $R(\widehat{s}_{\downarrow X},\widehat{s}'_{\downarrow X})$ by Definition 1 and $I(\widehat{s})\equiv T(\widehat{s}_{\downarrow\widehat{X}},\widehat{s}_{\downarrow X})$. Our goal is to show that $T(\widehat{s}'_{\downarrow\widehat{X}},\widehat{s}'_{\downarrow X})$ holds. By Definition 1 $\widehat{s}'_{\downarrow\widehat{X}}$ can be equal to

- $\widehat{s}_{\downarrow X}$ *due to transition of type (1).* From inductiveness of T and $R(\widehat{s}_{\downarrow X},\widehat{s}'_{\downarrow X})$, we get $T(\widehat{s}_{\downarrow X},\widehat{s}'_{\downarrow X})\equiv T(\widehat{s}'_{\downarrow\widehat{X}},\widehat{s}'_{\downarrow X})$.
- $\widehat{s}_{\downarrow\widehat{X}}$ *due to transition of type (2).* Again, from inductiveness of T, $T(\widehat{s}_{\downarrow\widehat{X}},\widehat{s}_{\downarrow X})$ and $R(\widehat{s}_{\downarrow X},\widehat{s}'_{\downarrow X})$, formula $T(\widehat{s}_{\downarrow\widehat{X}},\widehat{s}'_{\downarrow X})\equiv T(\widehat{s}'_{\downarrow\widehat{X}},\widehat{s}'_{\downarrow X})$ holds.

Direction $\Leftarrow$: Let I be an inductive 1-step invariant of P'. Let $s,s',s''\in\mathscr{R}$ be reachable states in P. To prove that T is an inductive transition invariant, we have to show two subgoals

- $R(s,s')\Rightarrow T(s,s')$. Since $R(s,s')$, then there exist states $\widehat{s},\widehat{s}'\in S'$ such that $R'(\widehat{s},\widehat{s}')$. Moreover, $\widehat{s}_{\downarrow X}=s$, $\widehat{s}'_{\downarrow X}=s'$ and we can save the state with $R'(\widehat{s},\widehat{s}')$. In more details, $\widehat{s}(saved)=0$ and $\widehat{s}'(saved)=1$, therefore $\widehat{s}'_{\downarrow\widehat{X}}=\widehat{s}_{\downarrow X}$. Note that since $s,s'\in\mathscr{R}$, then $\widehat{s},\widehat{s}'\in\mathscr{R}'$[3]. Finally, we get that $I(\widehat{s}')\equiv T(\widehat{s}'_{\downarrow\widehat{X}},\widehat{s}'_{\downarrow X})\equiv T(\widehat{s}_{\downarrow X},\widehat{s}'_{\downarrow X})\equiv T(s,s')$ holds.

[3] We omit the proof of this here. However, it follows from the construction in Definition 1.

- $T(s,s'') \wedge R(s'',s') \Rightarrow T(s,s')$. Due to $T(s,s'')$, there exists a state $\widehat{s} \in S'$ such that $\widehat{s}_{\downarrow \widehat{X}} = s$ and $\widehat{s}_{\downarrow X} = s''$, for which $I(\widehat{s})$ holds. Consider another state $\widehat{s}'$ such that $\widehat{s}'_{\downarrow \widehat{X}} = s$ and $\widehat{s}'_{\downarrow X} = s'$. Because of $R(s'',s')$, there exists a transition of type (1) $R'(\widehat{s},\widehat{s}')$. Since I is inductive and $I(\widehat{s})$, we obtain $I(\widehat{s}') \equiv T(s,s')$. □

D Proofs for Sect. 5

D.1 Proof of Lemma 5

Proof. Let R_L be a transition relation of loop L and let k be a number such that $1 \leq k$. Furter, let us assume that the base-case and step-case formulas for the k are valid. Consider an arbitrary sequence of reachable states $s'_0,...,s'_m \in \mathscr{R}$, with $1 \leq m$ and $R(s'_0,s'_1) \wedge ... \wedge R(s'_{m-1},s'_m)$ being valid. Since all the states $s'_0,...,s'_m$ are reachable, $\bigwedge_{0 \leq j \leq m} I_L(s_j)$ holds. To prove that T is a transition invariant, we have to show that $T(s'_0,s'_m)$ holds. We divide the proof into two cases:

- $m \leq k$: In this case, we get that formula $\forall s_0, \dots, s_m : R(s_0, s_1) \wedge ... \wedge R(s_{m-1}, s_m) \Rightarrow T(s_0, s_m)$ is valid due to the base case. If we substitute the universally quantifed states with $s'_0,...,s'_m$, we get that $T(s'_0,s'_m)$ holds.
- $m > k$: Since both m and k are natural numbers, one of the foundational results from number theory tells us that there are two numbers $1 \leq l$ and $0 \leq r < k$, such that $m = l \cdot k + r$. We adjust the equation to an equivalent one and assume that $0 \leq l$ and $1 \leq r \leq k$. After substituting formula from the base case, we get that $R(s'_0,s'_1) \wedge ... \wedge R(s'_{r-1},s'_r) \Rightarrow T(s'_0,s'_r)$, hence $T(s'_0,s'_r)$ holds. We can now substitute the step formula with $T(s'_0,s'_r) \wedge R(s'_r,s'_{r+1}) \wedge ... \wedge R(s'_{r+k-1},s'_{r+k}) \Rightarrow T(s'_0,s'_{r+k})$. Therefore, $T(s'_0,s'_{r+k})$ holds. After substituting the formula l-times, we get that $T(s'_0,s'_{l \cdot k+r}) \equiv T(s'_0,s'_m)$ holds.

□

D.2 Proof of Lemma 6

Proof. Let X be the set of variables that occur in T. Since all the variables from X have finite domains, then $S_{\downarrow X}$ is also finite. For contradiction, assume that T is not well-founded on the reachable states and `is_well_founded`(T,I_L) is valid. Then there exists an infinite sequence $s_0,s_1,... \in S$ such that $\forall 0 \leq i : T(s_i,s_{i+1}) \wedge (s_i,s_{i+1}) \in \mathscr{R} \times \mathscr{R}$. Moreover, it holds that $\forall 0 \leq i : I_L(s_i)$ because I_L is the conjunction of the supporting invariants for L. Since $S_{\downarrow X}$ is finite, there exist two states s,s' from the sequence, such that $s_{\downarrow X} = s'_{\downarrow X}$. Let $succ(s) = \{s'' \in S \mid T(s,s'') \wedge I_L(s'')\}$, we show by induction the following: If $\forall 1 \leq k : T(s_1,s_2) \wedge ... \wedge T(s_k,s_{k+1}) \wedge \forall 1 \leq i \leq k+1 : I_L(s_i)$, then there exist at least k states that are in $succ(s_1)$ but are not in $succ(s_k)$.

- *Base Case:* Let $k = 1$ and $T(s_1,s_2) \wedge I_L(s_1) \wedge I_L(s_2)$ hold. Since the formula `is_well_founded`(T,I_L) is valid, then by definition of $succ$, it holds that there is a state s_0 such that $s_0 \in succ(s_1)$ and $s_0 \notin succ(s_2)$. Moreover, due to the second clause of the conjunction, $succ(s_2) \subseteq succ(s_1)$.

- *Step Case:* Let $k>1$ and $T(s_1,s_2)\wedge\ldots\wedge T(s_k,s_{k+1})\wedge\forall 1\leq i\leq k+1: I_L(s_i)$ hold. It follows, from the induction hypothesis, that there are at least $k-1$ states that are in $succ(s_1)$ and not in $succ(s_k)$. Due to the validity of the formula $\texttt{is\_well\_founded}(T,I_L)$, it holds that $succ(s_{k+1})\subseteq succ(s_k)$ and therefore, these states are also not in $succ(s_{k+1})$. Moreover, there also exists a state s_0 that is in $succ(s_k)$ and not in $succ(s_{k+1})$. Hence, there exist at least k states that are in $succ(s_1)$ and not in $succ(s_{k+1})$.

Since T depends only on the variables from X, it holds that $succ(s)=succ(s')$. Therefore, the property that $succ(s)$ contains at least one state that is not in $succ(s')$, leads to a contradiction with $s_{\downarrow_X}=s'_{\downarrow_X}$. Hence, T must be well-founded on the reachable states. □

D.3 Proof of Lemma 7

Proof. Let there exist formulas $C_1,\ldots,C_n$ such that $T=C_1\vee\ldots\vee C_n$ and $\forall 1\leq i\leq n:\texttt{is\_disj\_well\_founded}(C_i,I_L)$. For the contradiction, let us assume that $R_L^+\cap(\mathscr{R}\times\mathscr{R})$ is not well-founded. Since $R_L^+\wedge(\mathscr{R}\times\mathscr{R})$ is not well-founded, there is exists an infinite sequence $s_0,s_1,\ldots\in\mathscr{R}$ such that $\forall 0\leq i: R_L^+(s_i,s_{i+1})$. Let X be the set of variables used in T with finite domains. Due to the infinite sequence satisfying $R_L^+\wedge(\mathscr{R}\times\mathscr{R})$, there exist two states in the sequence $s,s'\in\mathscr{R}$ such that $s_{\downarrow_X}=s'_{\downarrow_X}$ and $R_L^+(s,s')$. Since $R_L^+\Rightarrow T$, then trivially $R_L^+\cap(\mathscr{R}\times\mathscr{R})\Rightarrow T$ holds as well. Hence, $T(s,s')$ and $T(s,s)$ hold because of the finiteness of X. There is a formula C_i from the sequence $C_1,\ldots,C_n$, such that $C_i(s,s)$ which is a contradiction with $\texttt{is\_disj\_well\_founded}(C_i,I_L)$. □

D.4 Proof of Lemma 8

Proof. For the contradiction, let us assume that the overapproximating program is not terminating and T is well-founded. Hence, the transition relation R_T is not well-founded. Therefore, there exist an inifinite sequence $s_0,s_1,\ldots\in\mathscr{R}$, such that $\forall 0\leq i: LC(s_i)\wedge I_L(s_i)\wedge I_L(s_{i+1})\wedge T(s_i,s_{i+1})$. However, then T is also not well-founded, which is a contradiction. □