Borralleras, C. [et al.]. Proving termination through conditional termination. A: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. "Tools and Algorithms for the Construction and Analysis of Systems, 23rd International Conference, TACAS 2017: held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017: Uppsala, Sweden, April 22-29, 2017: proceedings, part I". Berlín: Springer, 2017, p. 99-117. The final authenticated version is available online at https://doi.org/10.1007/978-3-662-54577-5_6

Proving Termination through Conditional Termination*

Cristina Borralleras[1], Marc Brockschmidt[2], Daniel Larraz[3], Albert Oliveras[3], Enric Rodríguez-Carbonell[3], and Albert Rubio[3]

[1] Universitat de Vic - Universitat Central de Catalunya
[2] Microsoft Research, Cambridge
[3] Universitat Politècnica de Catalunya

Abstract. We present a constraint-based method for proving conditional termination of integer programs. Building on this, we construct a framework to prove (unconditional) program termination using a powerful mechanism to combine conditional termination proofs. Our key insight is that a conditional termination proof shows termination for a subset of program execution states which do not need to be considered in the remaining analysis. This facilitates more effective termination as well as non-termination analyses, and allows handling loops with different execution phases naturally. Moreover, our method can deal with sequences of loops compositionally. In an empirical evaluation, we show that our implementation VeryMax outperforms state-of-the-art tools on a range of standard benchmarks.

1 Introduction

Proving program termination requires not only synthesizing termination arguments, but also reasoning about reachability of program states, as most non-trivial programs contain subprocedures or loops that only terminate for the executions that actually reach them. Thus, a termination prover has to segment the program state space according to its termination behavior, ignoring non-terminating but unreachable states. Recent advances in termination proving try to tackle this problem by abducing conditions for non-termination, and focusing the termination proof search on the remaining state space [25,32]. However, these techniques rely on relatively weak non-termination proving techniques. Furthermore, different termination arguments may be required depending on how a loop or subprocedure is reached, and thus, even though no non-termination argument can be found, the state space needs to be segmented.

In this work, we propose to use preconditions for *termination* to drive the unconditional termination proof. The key insight is that a condition ϕ implying

* This work was partially supported by the project TIN2015-69175-C4-3-R (MINECO/FEDER) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement ERC-2014-CoG 648276 AUTAR).

termination allows a termination prover to focus on those program states in which $\neg\phi$ holds. To obtain preconditions for termination, we introduce a new constraint-based method that analyzes program components (i.e., loops or subprocedures) independently and synthesizes termination arguments together with a conditional supporting invariant [12]. To prove full program termination, we use a novel program transformation we call *unfolding* which syntactically splits terminating from potentially non-terminating states using the generated termination conditions. This allows us to combine several conditional termination arguments, each obtained for a small component of the program independently, into a proof for the input program. In summary, we present the following contributions:

- A new method based on Max-SMT for finding preconditions for termination (cf. Sect. 3 and Algo. 1).
- A framework to prove termination or non-termination by repeatedly simplifying the program analysis task by combining conditional termination arguments using the *unfolding* transformation (cf. Sect. 4 and Algo. 2).
- An implementation of the technique in our tool VeryMax for C++ input programs and an extensive experimental evaluation showing that it is not only more powerful than existing tools, but also more efficient (cf. Sect. 5).

2 Preliminaries

SAT, Max-SAT, and Max-SMT. Let $\mathcal{P}$ be a fixed set of *propositional variables*. For $p \in \mathcal{P}$, p and $\neg p$ are *literals*. A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$. A (CNF) *propositional formula* is a conjunction of clauses $C_1 \wedge \cdots \wedge C_m$. The problem of *propositional satisfiability (SAT)* is to determine whether a propositional formula F has a *model*, i.e., an assignment M that satisfies F, denoted by $M \models F$. An extension of SAT is *Satisfiability Modulo Theories (SMT)* [6], where one checks the satisfiability of a formula with literals from a given background theory. Another extension is *(weighted partial) Max-SAT* [6], where some clauses in the input formula are *soft clauses* with an assigned weight, and the others are *hard clauses*. Here, we look for a model of the hard clauses that maximizes the sum of the weights of the satisfied soft clauses. Finally, *Max-SMT* combines Max-SAT and SMT. In a *Max-SMT* problem a formula is of the form $H_1 \wedge \ldots \wedge H_n \wedge [S_1, \omega_1] \wedge \ldots \wedge [S_m, \omega_m]$, where the hard clauses H_i and the soft clauses S_j (with weight ω_j) are disjunctions of literals over a background theory, and the aim is to find a model of the hard clauses that maximizes the sum of the weights of the satisfied soft clauses.

Programs and States. We fix a set of integer program *variables* $\mathcal{V} = \{v_1, \ldots, v_n\}$ and denote by $F(\mathcal{V})$ the conjunctions of linear inequalities over the variables $\mathcal{V}$.

Let $\mathcal{L}$ be the set of program *locations*, which contains a *canonical initial location* ℓ_{init}. Program *transitions* are tuples (ℓ_s, ρ, ℓ_t), where ℓ_s and $\ell_t \in \mathcal{L}$ are the source and target locations respectively, and $\rho \in F(\mathcal{V} \cup \mathcal{V}')$ describes the transition relation. Here $\mathcal{V}' = \{v_1', \ldots, v_n'\}$ represent the values of the program variables after the transition.[4]

[4] For $\varphi \in F(\mathcal{V})$, we denote by $\varphi' \in F(\mathcal{V}')$ the version of φ using primed variables.

A *program* $\mathcal{P}$ is a set of transitions.[5] The set of locations in these transitions is denoted by $\mathcal{L}(\mathcal{P})$. We identify a program with its *control-flow graph* (CFG), a directed graph in which nodes are the locations and edges are the transitions.[6] A *program component* $\mathcal{C}$ of a program $\mathcal{P}$ is the set of transitions of a *strongly connected component* (SCC) of the CFG of $\mathcal{P}$. Its *entry transitions* $\mathcal{E}_\mathcal{C}$ are those transitions $\tau = (\ell_s, \rho, \ell_t)$ such that $\tau \notin \mathcal{C}$ but $\ell_t \in \mathcal{L}(\mathcal{C})$ (and in this case ℓ_t is called an *entry location*), while its *exit transitions* $\mathcal{X}_\mathcal{C}$ are such that $\tau \notin \mathcal{C}$ but $\ell_s \in \mathcal{L}(\mathcal{C})$ (and then ℓ_s is an *exit location*).

A *state* $s = (\ell, v)$ consists of a location $\ell \in \mathcal{L}$ and a *valuation* $v : \mathcal{V} \to \mathbb{Z}$. *Initial* states are of the form (ℓ_{init}, v). We denote a *computation step* with transition $\tau = (\ell_s, \rho, \ell_t)$ by $(\ell_s, v) \to_\tau (\ell_t, w)$, where $(v, w) \models \rho$. We use $\to_\mathcal{P}$ if we do not care about the executed transition of $\mathcal{P}$, and $\to^*_\mathcal{P}$ to denote the transitive-reflexive closure of $\to_\mathcal{P}$. Sequences of computation steps are called *computations*.

Safety and Termination. An *assertion* (ℓ, φ) is a pair of a location ℓ and a formula $\varphi \in F(\mathcal{V})$. A program $\mathcal{P}$ is *safe* for the assertion (ℓ, φ) if for every computation starting at an initial state s_0 of the form $s_0 \to^*_\mathcal{P} (\ell, v)$, we have that $v \models \varphi$ holds. Safety can be proved using *conditional invariants* [12], which like ordinary invariants are inductive, but not necessarily initiated in all computations.

Definition 1 (Conditional Inductive Invariant). *Let $\mathcal{P}$ be a program. We say a map $\mathcal{Q} : \mathcal{L}(\mathcal{P}) \to F(\mathcal{V})$ is a* conditional (inductive) invariant *for $\mathcal{P}$ if for all $(\ell_s, v) \to_\mathcal{P} (\ell_t, w)$, we have $v \models \mathcal{Q}(\ell_s)$ implies $w \models \mathcal{Q}(\ell_t)$.*

A program $\mathcal{P}$ is *terminating* if any computation starting at an initial state is finite. An important tool for proving termination are *ranking functions*:

Definition 2 (Ranking Function). *Let $\mathcal{C}$ be a component of a program $\mathcal{P}$, and $\tau = (\ell_s, \rho, \ell_t) \in \mathcal{C}$. A function $\mathcal{R} : \mathbb{Z}^n \to \mathbb{Z}$ is a* ranking function *for τ if:*

- $\rho \models \mathcal{R} \geq 0$ ***[Boundedness]***
- $\rho \models \mathcal{R} > \mathcal{R}'$ ***[Decrease]***

and for every $(\hat{\ell_s}, \hat{\rho}, \hat{\ell_t}) \in \mathcal{C}$,

- $\hat{\rho} \models \mathcal{R} \geq \mathcal{R}'$ ***[Non-increase]***

The key property of ranking functions is that if a transition admits one, then it cannot be executed infinitely.

A core concept in our approach is *conditional termination*, i.e., the notion that once a condition holds, a program is definitely terminating. As we make heavy use of the program's control flow graph structure, we introduce this concept as location-dependent.

[5] Hence in our programming model procedure calls are not allowed. Note however that programs with non-recursive calls can also be handled by inlining the calls.

[6] Since we label transitions only with conjunctions of linear inequalities, disjunctive conditions are represented using several transitions with the same source and target location. Thus, $\mathcal{P}$ is actually a multigraph.

```
int x, y, z;
x = nondet();
y = nondet();
z = nondet();
while (y ≥ 0 && z ≠ 0) {
  if (z ≥ 0) {
    x = x − z;
    y = y + x;
    z = z + 1;
  } else {
    y = y + z;
    z = z − 1;
} }
```

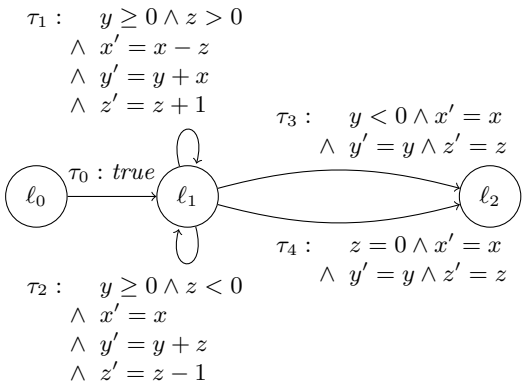

Fig. 1. Program and its CFG.

Definition 3 (Conditional Termination). *We say that a program $\mathcal{P}$ is* (ℓ, φ)-conditionally terminating *if every computation that contains a state (ℓ, v) with $v \models \varphi$ uses transitions from $\mathcal{P}$ only a finite number of times. In that case the assertion (ℓ, φ) is called a* precondition *for termination.*

3 Synthesizing Conditional Termination Arguments

Our approach for synthesizing conditional termination arguments works on one program component at a time. As proving that a program terminates is equivalent to showing that there is no program component where a computation can stay indefinitely, this turns out to be a convenient way to decompose termination proofs.

For a fixed program component $\mathcal{C}$, a conditional lexicographic termination argument is constructed iteratively by transition elimination as follows. In each iteration, we synthesize a linear ranking function together with supporting conditional invariants, requiring that they show that at least one transition of $\mathcal{C}$ is *finitely executable*, i.e., can only occur a finite number of times in any execution. The intuition here is that once we have proven that a transition τ can only be used finitely often, we only need to consider (possibly infinite) suffixes of program executions in which τ cannot appear anymore. If after some iterations no transition of $\mathcal{C}$ can be executed infinitely anymore, then the conjunction of all conditional invariants obtained at an entry location of $\mathcal{C}$ yields a precondition for termination. Indeed, once the conditional invariants hold at that entry location, then by inductiveness they hold from then on at all locations of $\mathcal{C}$, and hence the termination argument applies.

Example 1. Consider the program in Fig. 1 and its CFG, with initial location $\ell_{\text{init}} = \ell_0$. We want to find a precondition for termination of the component $\mathcal{C} = \{\tau_1, \tau_2\}$, corresponding to the **while** loop.

For $\tau = (\ell_s, \rho, \ell_t)$:

Initiation:	$\mathbb{I}_\tau$	$\stackrel{def}{=}$	ρ	$\Rightarrow I'_{\ell_t}$
Consecution:	$\mathbb{C}_\tau$	$\stackrel{def}{=}$	$I_{\ell_s} \wedge \rho$	$\Rightarrow I'_{\ell_t}$
Boundedness:	$\mathbb{B}_\tau$	$\stackrel{def}{=}$	$I_{\ell_s} \wedge \rho$	$\Rightarrow R \geq 0$
Decrease:	$\mathbb{D}_\tau$	$\stackrel{def}{=}$	$I_{\ell_s} \wedge \rho$	$\Rightarrow R > R'$
Non-increase:	$\mathbb{N}_\tau$	$\stackrel{def}{=}$	$I_{\ell_s} \wedge \rho$	$\Rightarrow R \geq R'$

Fig. 2. Constraints used for generating preconditions for termination.

In a first iteration, we generate the ranking function y for τ_2, together with the supporting conditional invariant $z < 0$. Note that $z < 0$ is indeed a conditional invariant: it is preserved by τ_2 as z decreases its value, and is also trivially preserved by τ_1 since this transition is in fact disabled if $z < 0$. Under the condition $z < 0$, y is bounded and decreases in τ_2, and τ_1 is disabled and so finitely executable. Hence, $(\ell_1, z < 0)$ is a precondition for termination. ■

As observed in [9, 30], synthesizing lexicographic termination arguments together with supporting invariants requires to keep several copies of the program under analysis. Thus, in the analysis of a component $\mathcal{C}$, we keep a set $\mathcal{M}$ of possibly infinitely executable transitions (i.e., those for which we have not proved conditional termination yet), called the *termination component*. Nonetheless, to compute sound invariants (i.e., soundly reason about reachable states), we need to take all transitions into account. However, these transitions can be strengthened with the supporting invariants that we synthesized in earlier proof steps. Hence, we keep another copy $\mathcal{I}$, called the *conditional invariant component*, which is like the original component $\mathcal{C}$, except for the addition of the conditional invariants found in previous iterations. Initially both the termination and the conditional invariant components are identical copies of the component $\mathcal{C}$.

The proposed method for generating preconditions for termination is an extension of the constraint-based approach for proving (unconditional) termination presented in [30]. The individual constraints used in our method are displayed in Fig. 2, corresponding to the standard constraints employed in constraint-based techniques [8]. For all locations ℓ in $\mathcal{C}$, we introduce templates I_ℓ corresponding to fixed-length conjunctions of linear inequalities on the program variables; i.e., I_ℓ is of the form $\bigwedge_{1 \leq i \leq k}(a_i + \sum_{v \in \mathcal{V}} a_{i,v} v \leq 0)$ for some k and where the a_* are integer template variables that do not appear in $\mathcal{V}$. Furthermore, we also define a template R for a linear ranking function[7] with integer coefficients, i.e., R is of the form $a + \sum_{v \in \mathcal{V}} a_v v$. For a given component $\mathcal{C}$ with entries $\mathcal{E}_\mathcal{C}$, we combine

[7] While using a different ranking function for each program location is possible, we have found that the added power does not justify the increased complexity of the ensuing SMT problem.

these constraints in the (non-linear) formula $\mathbb{F}$ as follows:

$$\mathbb{F} \stackrel{def}{=} \bigwedge_{\tau \in \mathcal{E}_\mathcal{C}} \mathbb{I}_\tau \wedge \bigwedge_{\tau \in \mathcal{I}} \mathbb{C}_\tau \wedge \bigwedge_{\tau \in \mathcal{M}} \mathbb{N}_\tau \wedge \bigvee_{\tau \in \mathcal{M}} (\mathbb{B}_\tau \wedge \mathbb{D}_\tau)\,.$$

However, not all of these constraints are treated as hard constraints. Most notably, we turn $\bigwedge_{\tau \in \mathcal{E}_\mathcal{C}} \mathbb{I}_\tau$ into soft constraints. Intuitively this means that, if possible, we want to synthesize a true (unconditional) supporting invariant, but will also allow invariants that do not always hold. However, we keep $\bigwedge_{\tau \in \mathcal{I}} \mathbb{C}_\tau$ as a hard constraint, ensuring that our conditional invariants are indeed inductive, i.e., keep on holding after they have been satisfied once. Similarly, $\bigwedge_{\tau \in \mathcal{M}} \mathbb{N}_\tau \wedge \bigvee_{\tau \in \mathcal{M}} (\mathbb{B}_\tau \wedge \mathbb{D}_\tau)$ are kept as hard constraints, enforcing that a true ranking function is found, though it may only hold in those cases where the supporting invariant is initiated. The conditions for the supporting invariants will eventually become our preconditions for termination.

Algo. 1 shows our procedure CondTerm for generating preconditions for termination. It takes as inputs the component $\mathcal{C}$ under consideration and its entry transitions $\mathcal{E}_\mathcal{C}$, and returns a conditional invariant $\mathcal{Q}$ that ensures that no infinite computation can remain within $\mathcal{C}$.

Algorithm 1 Procedure CondTerm for computing preconditions for termination

Input: component $\mathcal{C}$, entry transitions $\mathcal{E}_\mathcal{C}$
Output: None | $\mathcal{Q}$, where $\mathcal{Q}$ maps locations in $\mathcal{L}(\mathcal{C})$ to conjunctions of inequalities

1: $(\mathcal{I}, \mathcal{M}) \leftarrow (\mathcal{C}, \mathcal{C})$
2: $\mathcal{Q} \leftarrow \{\ \ell \mapsto \mathit{true} \mid \ell \in \mathcal{L}(\mathcal{C})\ \}$
3: **while** $\mathcal{M} \neq \varnothing$ **do**
4: construct formula $\mathbb{F}$ from $\mathcal{I}$, $\mathcal{M}$, $\mathcal{E}_\mathcal{C}$
5: $\sigma \leftarrow$ Max-SMT-solver($\mathbb{F}$)
6: **if** σ is a solution **then**
7: $\mathcal{I} \leftarrow \{\ (\ell_s,\ \rho \wedge \sigma(I_{\ell_s}),\ \ell_t)\ \mid\ (\ell_s, \rho, \ell_t) \in \mathcal{I}\ \}$
8: $\mathcal{M} \leftarrow \{\ (\ell_s,\ \rho \wedge \sigma(I_{\ell_s}),\ \ell_t)\ \mid\ (\ell_s, \rho, \ell_t) \in \mathcal{M}\ \}$
9: $\mathcal{M} \leftarrow \mathcal{M} - \{\ \tau \in \mathcal{M} \mid \sigma(R)$ is a ranking function for $\tau\ \}$
10: $\mathcal{Q} \leftarrow \{\ \ell \mapsto \mathcal{Q}(\ell) \wedge \sigma(I_\ell) \mid \ell \in \mathcal{L}(\mathcal{C})\ \}$
11: **else return** None
12: **return** $\mathcal{Q}$

In Algo. 1, we continue to extend the termination argument as long as there are still potentially infinitely executable transitions (line 3). For this, we build a Max-SMT problem $\mathbb{F}$ to generate a ranking function and its supporting conditional invariants. If no solution can be found, then the procedure gives up (line 11). Otherwise, a solution σ to $\mathbb{F}$ yields a linear function $\sigma(R)$ (the instantiation of the template ranking function R determined by σ) together with conditional invariants $\sigma(I_{\ell_s})$. Since the $\sigma(I_{\ell_s})$ are conditional invariants, they can be used to strengthen transitions $\tau = (\ell_s, \rho, \ell_t)$ by conjoining $\sigma(I_{\ell_s})$ to ρ, both in the conditional invariant component and in the termination component (lines 7-8).

Most importantly, we identify the subset of the transitions τ from $\mathcal{M}$ for which $\sigma(R)$ is a ranking function, and hence can be removed from $\mathcal{M}$ (line 9). Finally, conditional invariants from previous iterations are accumulated so that, in the end, a global conjunction can be returned (lines 10 and 12).

In essence, this process corresponds to the step-wise construction of a lexicographic termination argument. For a location ℓ at which the component $\mathcal{C}$ is entered, the conjunction of all obtained $\sigma(I_\ell)$ is then a precondition for termination. The following theorem states the correctness of procedure CondTerm:

Theorem 1 (CondTerm soundness). *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a component, and $\mathcal{E}_\mathcal{C}$ its entry transitions. If the procedure call* $\mathsf{CondTerm}(\mathcal{C}, \mathcal{E}_\mathcal{C})$ *returns $\mathcal{Q} \neq \mathsf{None}$, then $\mathcal{C}$ is $(\ell, \mathcal{Q}(\ell))$-conditionally terminating for any $\ell \in \mathcal{L}(\mathcal{C})$.*

Of course, Algo. 1 is an idealized, high-level description of our procedure. In an implementation of the procedure CondTerm, a number of small changes help to improve the overall number of solved instances.

Constraint Strengthening. Additional constraints can be added to formula $\mathbb{F}$ to favor conditional invariants that are more likely to be useful. In particular, a constraint requiring that the conditional invariants are compatible with the entry transitions and with the previously generated conditional invariants has proven useful, i.e.

$$\bigvee_{(\ell_s,\, \rho,\, \ell_t) \in \mathcal{E}_\mathcal{C}} \exists \mathcal{V}, \mathcal{V}' \; (\, I'_{\ell_t} \wedge \rho \wedge \mathcal{Q}(\ell_t)' \,) \, .$$

Constraint Softening. Similarly, we can increase the allowed range of models by turning more of the clauses into soft clauses. For example, this can be used to allow *quasi-ranking functions* [30] in addition to ranking functions. Quasi-ranking functions are functions that satisfy the non-increase condition, but may fail to decrease or be bounded, or both. By using them to partition transitions and perform case analysis, programs can also be shown to be terminating.

Pseudo-Invariants of Termination Component. In some circumstances, inductive properties of the termination component $\mathcal{M}$ (i.e., satisfying **Consecution** only for transitions in $\mathcal{M}$) can be sound and useful; namely, when the complement of the property disables a transition.

Formally, let $\mathcal{Q}$ be a map from $\mathcal{L}(\mathcal{P})$ to $F(\mathcal{V})$ such that $\mathcal{Q}(\tilde{\ell_s}) \wedge \tilde{\rho} \Rightarrow \mathcal{Q}(\tilde{\ell_t})'$ for all $(\tilde{\ell_s}, \tilde{\rho}, \tilde{\ell_t}) \in \mathcal{M}$, and $\neg \, \mathcal{Q}(\ell_s) \wedge \rho \models \mathit{false}$ for some $\tau = (\ell_s, \rho, \ell_t) \in \mathcal{M}$. Moreover, assume that $\mathcal{Q}$ supports a ranking function $\mathcal{R}$ for τ. Then τ can only be used finitely often. To see this, assume that there is an infinite computation in which τ occurs infinitely often. Then there is a state at location ℓ_s in the computation from which only transitions in $\mathcal{M}$ are taken. Since $\mathcal{Q}$ is inductive over transitions in $\mathcal{M}$, if $\mathcal{Q}(\ell_s)$ holds at that state then it holds from then on, and therefore $\mathcal{R}$ proves that τ cannot be executed an infinite number of times. Otherwise, if $\mathcal{Q}(\ell_s)$ does not hold, then τ cannot be executed at all. This weaker requirement on $\mathcal{Q}$ allows removing transitions from $\mathcal{M}$ and is easier to satisfy. Still, it is insufficient to do a case analysis as a full conditional invariant allows.

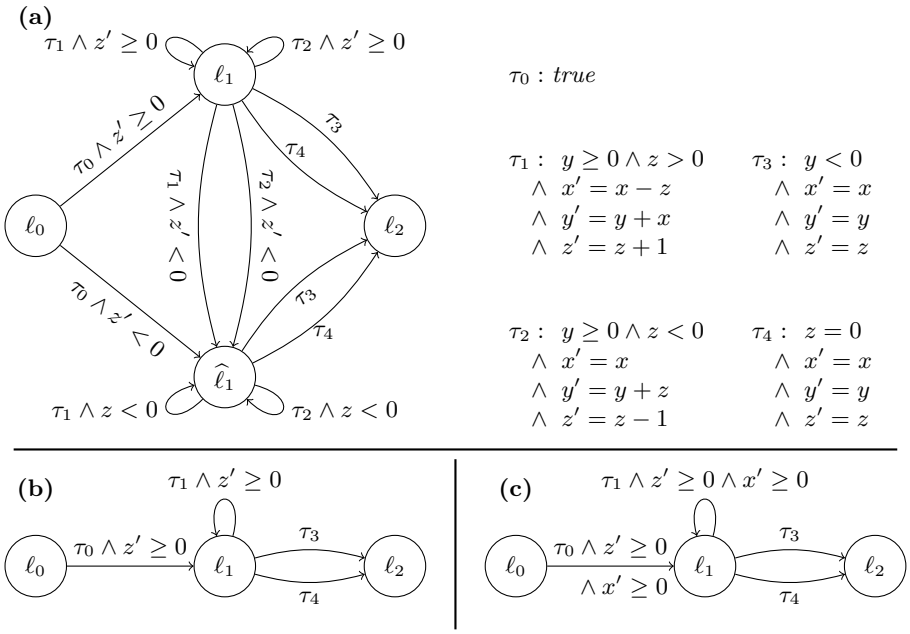

Fig. 3. Unfolding of the program from Fig. 1 for conditional invariant $z < 0$ at ℓ_1 (**a**), ensuing narrowing/simplification (**b**) and narrowing after unfolding for $x < 0$ at ℓ_1 (**c**).

4 Proving Termination using Conditional Termination

Our key contribution is to leverage conditional termination arguments to perform a natural case analysis of program executions. In this way, as our analysis progresses, more and more program runs are excluded from the program analysis, allowing the method to focus on those parts of the program for which termination has not been guaranteed yet. The core component of this is a syntactic program transformation we call *unfolding* that implements the semantic intuition of distinguishing program states for which termination has not been proven yet.

4.1 Program Unfoldings

We begin this subsection with an example that illustrates how conditional invariants can be used to unfold the component under consideration.

Example 2. Consider the program from Fig. 1 again. In Ex. 1 it was shown that all computations for which $z < 0$ holds at location ℓ_1 are finite. In fact, a byproduct of the proof was that $z < 0$ is a conditional invariant at location ℓ_1. We show how to exploit this to prove *unconditional* termination next.

Following the intuition of a case analysis, we unfold the program component by introducing a copy of it in which we assume that the conditional invariant

holds. In our example, we duplicate the location ℓ_1, introducing a copy denoted by $\widehat{\ell_1}$. We also duplicate all transitions in, from and to the component, using the newly introduced location. However, all copied transitions should also reflect our case analysis, and are thus strengthened by the conditional invariant $z < 0$. In our case analysis, the original component now corresponds to the case that the conditional invariant does *not* hold, and thus, all of the original transitions are strengthened to assume the negation of the conditional invariant. Finally, to allow for computations where the invariant eventually becomes true, we add copies of the transitions from the original component to the copied location, again strengthened by the invariant. The resulting program is shown in Fig. 3(a).

The original program and its unfolding behave equivalently, in particular regarding termination. However, we already know from Ex. 1 that under the assumption $z < 0$, the new component has no infinite executions. Hence, we can *narrow* the set of potentially infinite computations and focus on the program shown in Fig. 3(b), obtained by removing all known-terminating locations (i.e., $\widehat{\ell_1}$) from the unfolding and simplifying. If this narrowed program terminates, we can conclude that the original program terminates too.

Synthesizing another conditional termination argument for the program from Fig. 3(b) now yields the ranking function y, supported by the conditional invariant $x < 0$ at ℓ_1. Then we can unfold with $x < 0$ again and narrow, obtaining the program in Fig. 3(c). Finally this program can be proven terminating with ranking function x without the need of any conditional invariant and, hence, without precondition. This concludes the proof of termination of the original program.

Note that the unfolding/narrowing mechanism provides not only a termination proof but also a characterization of the program execution phases. In particular, our example can be viewed to have three phases, corresponding to the unfoldings we have applied. One phase corresponds to the case where $z < 0$ (where the `else`-block is repeatedly used), one to the case $z > 0 \wedge x < 0$, and finally, one corresponds to the case $z > 0 \wedge x \geq 0$. ■

To formalize this execution phase-structured proof technique, we first define the *unfolding* program transformation:

Definition 4. *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a component of $\mathcal{P}$, $\mathcal{E}_\mathcal{C}$ its entry transitions, $\mathcal{X}_\mathcal{C}$ its exit transitions, and $\mathcal{Q} : \mathcal{L}(\mathcal{C}) \to F(\mathcal{V})$ a conditional invariant for $\mathcal{C}$. The* unfolding *of $\mathcal{P}$ is*

$$
\begin{aligned}
\widehat{\mathcal{P}} = \quad & \{\ (\ell_s,\ \rho \wedge \neg\ \mathcal{Q}(\ell_t)',\ \ell_t), \\
& \quad\ (\ell_s,\ \rho \wedge\ \ \mathcal{Q}(\ell_t)',\ \widehat{\ell_t}), \\
& \quad\ (\widehat{\ell_s},\ \rho \wedge\ \ \mathcal{Q}(\ell_s),\ \ \widehat{\ell_t}) \mid (\ell_s, \rho, \ell_t) \in \mathcal{C}\ \} \\
& \cup \{\ (\ell_s,\ \rho \wedge \neg \mathcal{Q}(\ell_t)',\ \ell_t), \\
& \quad\ (\ell_s,\ \rho \wedge\ \ \mathcal{Q}(\ell_t)',\ \widehat{\ell_t}) \mid (\ell_s, \rho, \ell_t) \in \mathcal{E}_\mathcal{C}\ \} \\
& \cup \{\ (\ell_s,\ \rho, \ell_t), \\
& \quad\ (\widehat{\ell_s},\ \rho, \ell_t)\ \mid\ (\ell_s, \rho, \ell_t) \in \mathcal{X}_\mathcal{C}\ \} \\
& \cup \{\tau\ \mid\ \tau \in \mathcal{P} \setminus (\mathcal{C} \cup \mathcal{E}_\mathcal{C} \cup \mathcal{X}_\mathcal{C})\}
\end{aligned}
$$

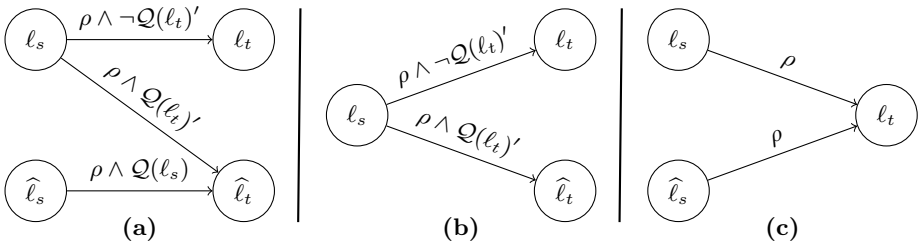

Fig. 4. Transitions in unfolding $\widehat{\mathcal{P}}$ for conditional invariant $\mathcal{Q}$ corresponding to a transition $\tau = (\ell_s, \rho, \ell_t) \in \mathcal{P}$, depending on whether (**a**) $\tau \in \mathcal{C}$, (**b**) $\tau \in \mathcal{E}_\mathcal{C}$, (**c**) $\tau \in \mathcal{X}_\mathcal{C}$.

where for each $\ell \in \mathcal{L}(\mathcal{C})$ there is a fresh location $\widehat{\ell}$ such that $\widehat{\ell} \notin \mathcal{L}(\mathcal{P})$.

Fig. 4 represents graphically how a transition of the original program is transformed, depending on whether it is a transition of the component, an entry transition, or an exit transition. The following result states that a program and its unfolding are semantically equivalent, i.e., that the encoded case analysis is complete.

Theorem 2. *Given states (ℓ_0, v_0) and (ℓ_k, v_k) such that $\ell_0, \ell_k \in \mathcal{L}(\mathcal{P})$, there is a computation in $\mathcal{P}$ of length k of the form $(\ell_0, v_0) \to^*_\mathcal{P} (\ell_k, v_k)$ if and only if there is a computation in $\widehat{\mathcal{P}}$ of length k of the form $(\ell_0, v_0) \to^*_{\widehat{\mathcal{P}}} (\ell_k, v_k)$ or of the form $(\ell_0, v_0) \to^*_{\widehat{\mathcal{P}}} (\widehat{\ell}_k, v_k)$.*

Now we are ready to formally define the narrowing of a program:

Definition 5. *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a component of $\mathcal{P}$, $\mathcal{E}_\mathcal{C}$ its entry transitions, and $\mathcal{Q} : \mathcal{L}(\mathcal{C}) \to F(\mathcal{V})$ a conditional invariant for $\mathcal{C}$. The* narrowing *of $\mathcal{P}$ is:*

$$narrow(\mathcal{P}) = \{\ (\ell_s,\ \rho \wedge \neg\ \mathcal{Q}(\ell_t)',\ \ell_t)\ \mid\ (\ell_s, \rho, \ell_t) \in \mathcal{C} \cup \mathcal{E}_\mathcal{C}\ \} \cup \{\tau\ \mid\ \tau \in \mathcal{P} - (\mathcal{C} \cup \mathcal{E}_\mathcal{C})\}$$

The narrowing of a program can be viewed as the result of eliminating the copies $\widehat{\ell}$ of the locations $\ell \in \mathcal{L}(\mathcal{C})$ in the unfolding $\widehat{\mathcal{P}}$ and their corresponding transitions. Alternatively, one may take the original program $\mathcal{P}$ and consider that for any transition $\tau = (\ell_s, \rho, \ell_t) \in \mathcal{C} \cup \mathcal{E}_\mathcal{C}$, the relation is replaced by $\rho \wedge \neg\, \mathcal{Q}(\ell_t)'$. The intuition is that, if a call to $\mathsf{CondTerm}(\mathcal{C}, \mathcal{E}_\mathcal{C})$ has been successful (i.e., $\mathcal{Q} \stackrel{def}{=} \mathsf{CondTerm}(\mathcal{C}, \mathcal{E}_\mathcal{C}) \neq \mathsf{None}$), by the inductiveness of $\mathcal{Q}$ a computation that satisfies $\mathcal{Q}(\ell)$ for a certain $\ell \in \mathcal{C}$ cannot remain within $\mathcal{C}$ indefinitely. Hence we only need to consider computations such that whenever a location $\ell \in \mathcal{C}$ is reached, we have that $\mathcal{Q}(\ell)$ does not hold.

Corollary 1. *Let $\mathcal{P}$ be a program, $\mathcal{C}$ a component of $\mathcal{P}$, $\mathcal{E}_\mathcal{C}$ its entry transitions, and $\mathcal{Q} : \mathcal{L}(\mathcal{C}) \to F(\mathcal{V})$ a conditional invariant for $\mathcal{C}$ obtained from a call to $\mathsf{CondTerm}(\mathcal{C}, \mathcal{E}_\mathcal{C})$. There is an infinite computation in $\mathcal{P}$ that eventually only uses*

transitions from $\mathcal{C}$ if and only if there is such a computation in narrow($\mathcal{P}$) using transitions from narrow($\mathcal{C}$).

Proof. The right to left implication holds by Thm. 2 as narrow($\mathcal{P}$) $\subseteq \widehat{\mathcal{P}}$. For the left to right implication, by Thm. 2 an infinite computation of $\mathcal{P}$ staying in $\mathcal{C}$ yields an infinite computation of $\widehat{\mathcal{P}}$ staying in $\widehat{\mathcal{C}}$. By induction, for any location $\ell \in \mathcal{L}(\mathcal{C})$ we have that $\mathcal{Q}(\ell)$ is an (unconditional) invariant at location $\widehat{\ell}$ of $\widehat{\mathcal{P}}$: now the initiation condition also holds by definition because all transitions arriving at $\widehat{\ell}$ require $\mathcal{Q}(\ell)$ to hold. By Thm. 1, no infinite computation in $\widehat{\mathcal{P}}$ staying in $\widehat{\mathcal{C}}$ can reach a location of the form $\widehat{\ell}$, where $\ell \in \mathcal{L}(\mathcal{C})$. So such an infinite computation is a computation of narrow($\mathcal{P}$) that eventually only uses transitions from narrow($\mathcal{C}$). □

Our termination proofs are sequences of relatively simple program transformations and termination proving techniques. This formal simplicity allows one to easily implement, extend and certify the technique. As discussed in Ex. 2, the unfolding/narrowing mechanism provides not only a termination proof, but also a characterization of the execution phases. In contrast to other works [37], these phases are obtained semantically from the generated conditional invariants, and do not require syntactic heuristics.

4.2 Proving Program Termination

So far we have discussed the handling of a single component at a time. By combining our method with an off-the-shelf safety checker, full program termination can be proven too. The next example illustrates this while comparing with previous Max-SMT-based techniques for proving termination [30].

Example 3. The method from [30] considers components following a topological ordering. Each component is dealt with locally: only its transitions and entries are taken into account, independently of the rest of the program. The analysis of a component concludes when the component is proven (unconditionally) terminating. Hence, for the program in Fig. 5, the first loop is proven terminating using the ranking function z. However, if no additional information is inferred, then the proof of termination of the second loop cannot succeed: the necessary invariant that $x \geq 1$ between the two loops, at program location ℓ, is missing.

```
int x = nondet();
int y = nondet();
int z = nondet();
assume(x > z && z ≥ 0);
while (z > 0) {
  x = x − 1;
  z = z − 1; }
ℓ : while (y < 0)
  y = y + x;
```

Fig. 5. Program that cannot be proven terminating with the approach in [30].

On the other hand, the approach proposed here is able to handle this program successfully. Indeed, the first loop can be proven terminating with z as a ranking function as observed above. Regarding the second loop, the conditional invariant $x \geq 1$ together with the ranking function $-y$ are generated. To prove $x \geq 1$ holds at ℓ a safety checker may be used, which then makes a global analysis to verify

Algorithm 2 Procedure Term for proving or disproving program termination

Input: program $\mathcal{P}$
Output: Yes (resp., No) if $\mathcal{P}$ terminates (resp., does not terminate), or $\perp$ if unknown

```
 1: S ← stack of components of P
 2: while S ≠ ∅ ∧ ¬timed_out() do
 3:     C ← Pop(S)
 4:     E ← E_C
 5:     Q ← CondTerm(C, E)
 6:     while Q ≠ None do
 7:         if ∀ℓ∈L(C)∩L(E): P is safe for assertion(ℓ, Q(ℓ)) then {Call safety check}
 8:             break
 9:         C ← { (ℓ_s, ρ ∧ ¬Q(ℓ_t)', ℓ_t) | (ℓ_s, ρ, ℓ_t) ∈ C }      {Narrow component}
10:         E ← { (ℓ_s, ρ ∧ ¬Q(ℓ_t)', ℓ_t) | (ℓ_s, ρ, ℓ_t) ∈ E }        {Narrow entries}
11:         Q ← CondTerm(C, E)
12:     if Q = None then return (ProvedNonTermination(C, E, P) ? No : ⊥)
13: return (S = ∅ ? Yes : ⊥)
```

the truth of the assertion. Finally full termination can be established. Note that components may be considered in any order, not necessarily a topological one. ■

Ex. 3 illustrates that combining our conditional termination proving technique with a safety checker is necessary to efficiently handle long and complex programs.

It is also important to note that, as the proof of Corollary 1 indicates, the narrowed program is termination-equivalent to the original program. In particular, this means that a non-termination proof for the narrowed program is a non-termination proof for the original program, as only terminating computations have been discarded by the transformation. Further, our program transformation does not only add information to the entry transitions (as in [32]) but also to all transitions occurring in the component under analysis. This significantly improves the precision of our otherwise unchanged non-termination analysis (cf. Sect. 5).

Altogether, our procedure for proving or disproving program termination is described in Algo. 2. It takes as input a program $\mathcal{P}$ and returns Yes if the program can be proved to terminate, No if it can be proved not to terminate, or $\perp$ otherwise. Components are handled in sequence one at a time provided the time limit has not been exceeded (line 2). For each component, preconditions for termination are computed (lines 5 and 11), which are then checked to hold by calling an external safety checker (line 7). If this test is passed, the component is guaranteed to terminate and the next one can be considered. Otherwise narrowing is applied (lines 9-10) and the process is repeated. If at some point the generation of preconditions for termination fails, then non-termination is attempted by calling an out-of-the-box non-termination prover (line 12). Note that the outer loop can easily be parallelized (i.e., all components can be considered at the same time), and that similarly, the generation of more preconditions can be attempted in parallel to the safety checks for already generated termination conditions. The correctness of Algo. 2 follows directly from Corollary 1.

5 Related Work & Experimental Results

We build on a rich tradition of methods to prove termination [1, 10, 16, 18, 21, 22, 25, 26, 28, 32, 36, 39, 42–44] and non-termination [4, 13, 14, 24, 29, 46] of imperative programs. Most notably, our constraint-based approach to conditional termination is an extension of existing work on ranking function synthesis using constraint solvers [2, 3, 5, 8, 23, 30, 33, 35], and is most closely related to our earlier work on using Max-SMT solving to infer quasi-ranking functions [30]. There, an independent invariant generation procedure was used before unconditional termination arguments were synthesized for program components. Thus, invariants were not generated "on demand" and the method fails on examples such as Ex. 3.

The key contribution of our method is to use conditional termination arguments to segment the state space for the remainder of the analysis. A related idea was used in TRex [25] and HipTNT+ [32], which alternate termination and non-termination proving techniques in their proof search. However, both approaches only use preconditions for *non-termination* to segment the state space, and thus are reliant on non-termination techniques. As finding non-termination arguments is an $\exists\forall\exists$ problem (there exists a state set such that for all its states there exists a computation leading back to it), these methods tend to be significantly weaker in practice than those based on termination, which is an $\exists\forall$ problem (there exists a ranking function such that all computations decrease it).

A related idea is counterexample-guided termination proving [10, 16, 26, 28, 36], in which a speculated termination argument is refined until it covers all program executions. Thus, these methods *grow* a set of terminating program states, whereas our method *shrinks* the set of potentially non-terminating states. In practice, "ignoring" the terminating states in a safety prover is often a non-trivial semantic operation, in contrast to the effects of our syntactic *narrowing* operation.

Proving conditional termination has seen less interest than full termination analysis. A first technique combined constraint-based methods for finding potential ranking functions with quantifier elimination [15] to infer preconditions. More recently, policy iteration-based methods [34], backwards reasoning in the abstract interpretation framework [45] and an adaptation of conflict-driven learning from satisfiability solving [17] have been adapted to find conditions for termination. Our algorithm CondTerm differs in its relative simplicity (by delegating the majority of the work to a constraint solver), and our procedure Term could combine it with or replace it by other approaches. Finally, in a related line of work, decision procedures for conditional termination on restricted programming languages (e.g., only using linear or affine operations) have been developed [7].

Evaluation To evaluate our method, we have implemented Algo. 2 in the tool VeryMax, using it as a safety prover [12] and a non-termination prover [29]. The effectiveness of VeryMax depends crucially on the underlying non-linear Max-SMT solver, described in [31]. All experiments were carried out on the StarExec cluster [40], whose nodes are equipped with Intel Xeon 2.4GHz processors. [8] We

[8] A binary of VeryMax as well as the detailed results of the experiments can be found at `http://www.cs.upc.edu/~albert/VeryMax.html`.

have compared VeryMax with a range of competing termination provers on three benchmark sets. The first two example sets are the benchmark suites Integer Transition Systems and C Integer used in termCOMP 2016 [41], on which we compare with a superset of the tools [22, 26, 27, 32, 44] that competed in these categories in the 2016 edition[9]. Following the rules of the competition, we use a wall clock timeout of 300 seconds for the C Integer benchmark set, and a wall clock timeout of 30 seconds for the Integer Transition Systems benchmark set. The results of these experiments are displayed in Tab. 1, where the "Term" (resp. "NTerm") column indicates the number of examples proven terminating (resp. non-terminating), "Fail" any kind of prover failure, and "TO" the number of times the timeout was reached. Finally, "Total (s)" indicates the total time spent on all examples, in seconds.

	C Integer					Integer Transition Systems				
Tool	Term	NTerm	Fail	TO	Total (s)	Term	NTerm	Fail	TO	Total (s)
AProVE	210	73	39	13	7547.68	623	386	11	202	9651.05
Ctrl	–	–	–	–	–	348	0	421	453	17229.10
HipTNT+	210	95	25	5	2615.80	–	–	–	–	–
SeaHorn	171	73	19	72	22499.33	–	–	–	–	–
Ultimate	208	98	23	6	4745.79	–	–	–	–	–
VeryMax	**213**	**100**	**22**	**0**	**2354.94**	**620**	**412**	**103**	**87**	**8481.39**

Table 1. Experimental results on benchmarks from termCOMP 2016.

We additionally evaluated our tool on the examples from the Termination category of the Software Verification Competition 2016, comparing to the participants in 2016 [22, 26, 44] with a CPU timeout of 900 seconds. As VeryMax has no support for recursion and pointers at the moment, we removed 273 examples using these features and tested the tools on the remaining 358 examples. The results of this experiment are shown in Tab. 2. Altogether, the overall experimental results show that our method is not only the most powerful combined termination and non-termination prover, but also more efficient than all competing tools.

Tool	Term	NTerm	Fail	TO	Total (s)
AProVE	222	76	41	19	10235.44
SeaHorn	189	75	22	72	34760.69
Ultimate	224	103	25	6	7882.13
VeryMax	**231**	**101**	**26**	**0**	**2444.29**

Table 2. Results on SV-COMP benchmarks.

Moreover, to analyze the effect of our narrowing technique on non-termination proofs, we experimented with the Integer Transition Systems benchmark set. Namely, in Tab. 3 we compare the performance of VeryMax when trying to prove non-termination

	NTerm	Exclusive
Narrowing	412	82
Original	334	4

Table 3. Impact of narrowing on non-termination proofs.

[9] Due to incompatibilities of input formats, some tools could not be run on some of the benchmark sets. This is indicated in the tables with a dash −.

of narrowed components (Narrowing row) against using the original component (Original row). For each case, column "NTerm" indicates the examples proven non-terminating, and column "Exclusive" identifies those that could only be proven with that approach. The results show that removing (conditionally) terminating computations from the analysis significantly improves the effectiveness of the analysis. Still, the more complex narrowed components make the non-termination procedure in VeryMax time out in 4 cases that are otherwise proved non-terminating when using the original program components.

Finally, we studied the gain obtained with constraint strengthening, constraint softening and pseudo-invariants of the termination component (see Section 3). While constraint softening and pseudo-invariants help in proving termination in few cases at the cost of a time overhead, constraint strengthening significantly improves both the number of problems proved terminating and the time required to do so.

6 Conclusions and Future Work

We have proposed a new method for modular termination proofs of integer programs. A program is decomposed into program components, and conditional termination arguments are sought for each component separately. Termination arguments are synthesized iteratively using a template-based approach with a Max-SMT solver as a constraint solving engine. At each iteration, conditional invariants and ranking functions are generated which prove termination for a subset of program execution states. The key step of our technique is to exclude these states from the remaining termination analysis. This is achieved by narrowing, i.e., strengthening the transitions of the component and its entry transitions with the negation of the conditional invariant. This operation of narrowing can be viewed as unfolding the program in two phases, namely when the conditional termination argument holds and when it does not, and focusing on the latter, for which termination is not guaranteed yet.

In the future, we want to remove some of the limitations of our method. For example, we do not support the heap at this time, and combining our conditional termination proving procedure with a heap analysis would greatly extend the applicability of our approach. Moreover, as in many other techniques, numbers are treated as mathematical integers, not machine integers. However, a transformation that handles machine integers correctly by inserting explicit normalization steps at possible overflows [19] could be added. We are also interested in formally verifying our technique and to produce certificates for termination that can be checked by theorem provers [11]. Finally, we plan to extend our technique to proving bounds on program complexity. Finding such bounds is closely related to termination proving, and also requires to distinguish different phases of the execution precisely [20, 38]. Our termination proving method does this naturally, and an adaption to complexity could thus yield more precise bounds.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java Bytecode. In *FMOODS*, 2008.
2. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, 2010.
3. R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *IC*, 215, 2012.
4. A. Bakhirkin and N. Piterman. Finding recurrent sets with backward analysis and trace partitioning. In *TACAS*, 2016.
5. A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. In *POPL*, 2013.
6. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
7. M. Bozga, R. Iosif, and F. Konecný. Deciding conditional termination. *LCMS*, 10(3), 2014.
8. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, 2005.
9. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *CAV*, 2013.
10. M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: temporal property verification. In *TACAS*, 2016.
11. M. Brockschmidt, S. J. Joosten, R. Thiemann, and A. Yamada. Certifying safety and termination proofs for integer transition systems. In *WST*, 2016.
12. M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Compositional safety verification with max-smt. In *FMCAD*, 2015.
13. M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and `NullPointerExceptions` for JBC. In *FoVeOOS*, 2011.
14. H. Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. Proving nontermination via safety. In *TACAS*, 2014.
15. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *CAV*, 2008.
16. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
17. V. D'Silva and C. Urban. Conflict-driven conditional termination. In *CAV*, 2015.
18. S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *RTA*, 2011.
19. S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, 2012.
20. A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *APLAS*, pages 275–295, 2014.
21. P. Ganty and S. Genaim. Proving termination starting from the end. In *CAV*, 2013.
22. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *JAR*, 2016. To appear.
23. L. Gonnord, D. Monniaux, and G. Radanne. Synthesis of ranking functions using extremal counterexamples. In *PLDI*, 2015.
24. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*, 2008.

25. W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for termination. In *SAS*, 2010.
26. M. Heizmann, J. Hoenicke, and A. Podelski. Termination analysis by learning terminating programs. In *CAV*, 2014.
27. C. Kop and N. Nishida. Constrained term rewriting tool. In *LPAR-20*, 2015.
28. D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV*, 2009.
29. D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving non-termination using Max-SMT. In *CAV*, 2014.
30. D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD*, 2013.
31. D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Minimal-model-guided approaches to solving polynomial constraints and extensions. In *SAT*, 2014.
32. T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *PLDI*, 2015.
33. J. Leike and M. Heizmann. Ranking templates for linear loops. In *TACAS*, 2014.
34. D. Massé. Policy iteration-based conditional termination and ranking functions. In *VMCAI*, 2014.
35. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.
36. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
37. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, 2011.
38. M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, pages 745–761, 2014.
39. F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *TOPLAS*, 32(3), 2010.
40. A. Stump, G. Sutcliffe, and C. Tinelli. Starexec: a cross-community infrastructure for logic solving. In *IJCAR*, 2014.
41. Termination Competition. `http://termination-portal.org/wiki/Termination_Competition`.
42. A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop summarization and termination analysis. In *TACAS*, 2011.
43. C. Urban. The abstract domain of segmented ranking functions. In *SAS*, 2013.
44. C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing ranking functions from bits and pieces. In *TACAS*, 2016.
45. C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. In *SAS*, 2014.
46. H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In *TAP*, 2008.