6-2021

Proving non-termination by program reversal

Krishnendu CHATTERJEE

Ehsan Kafshdar GOHARSHADY

Petr NOVOTNÝ

Dorde ZIKELIC
Singapore Management University, dzikelic@smu.edu.sg

Proving Non-termination by Program Reversal

Krishnendu Chatterjee
IST Austria
Klosterneuburg, Austria
krishnendu.chatterjee@ist.ac.at

Ehsan Kafshdar Goharshady
Ferdowsi University of Mashhad
Mashhad, Iran
e.goharshady1@gmail.com

Petr Novotný
Masaryk University
Brno, Czech Republic
petr.novotny@fi.muni.cz

Đorđe Žikelić
IST Austria
Klosterneuburg, Austria
dzikelic@ist.ac.at

Abstract

We present a new approach to proving non-termination of non-deterministic integer programs. Our technique is rather simple but efficient. It relies on a purely syntactic reversal of the program's transition system followed by a constraint-based invariant synthesis with constraints coming from both the original and the reversed transition system. The latter task is performed by a simple call to an off-the-shelf SMT-solver, which allows us to leverage the latest advances in SMT-solving. Moreover, our method offers a combination of features not present (as a whole) in previous approaches: it handles programs with non-determinism, provides relative completeness guarantees and supports programs with polynomial arithmetic. The experiments performed with our prototype tool RevTerm show that our approach, despite its simplicity and stronger theoretical guarantees, is at least on par with the state-of-the-art tools, often achieving a non-trivial improvement under a proper configuration of its parameters.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Software verification**.

Keywords: Static Analysis, Program Termination, Backward Analysis, Invariant Generation, Completeness Guarantees

ACM Reference Format:
Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2021. Proving Non-termination by Program Reversal. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454093

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454093

1 Introduction

Program analysis. There are two relevant directions in program analysis: to prove program correctness and to find bugs. While a correctness proof is obtained once, the procedure of bug finding is more relevant during software development and is repeatedly applied, even for incomplete or partial programs. In terms of specifications, the most basic properties in program analysis are safety and liveness.

Program analysis for safety and termination. The analysis of programs with respect to safety properties has received a lot of attention [2, 27, 28, 33], and for safety properties to report errors the witnesses are finite traces violating the safety property. The most basic liveness property is termination. There is a huge body of work for proving correctness with respect to the termination property [7, 17, 21, 23], e.g. sound and complete methods based on ranking functions have been developed [16, 40, 41], and efficient computational approaches based on lexicographic ranking functions have also been considered [7, 8, 20].

Proving non-termination. The bug finding problem for the termination property, or proving non-termination, is a challenging problem. Conceptually, while for a safety property the violating witness is a finite trace, for a termination property the violating witnesses are infinite traces. There are several approaches for proving non-termination; here we discuss some key ones, which are most related in spirit to our new method (for a detailed discussion of related work, see Section 7). For the purpose of this overview, we (rather broadly and with a certain grain of salt) classify the approaches into two categories: *trace-based* approaches, which look for a non-terminating trace (e.g. [24, 30, 39]), and *set-based approaches,* which look for a set of non-terminal program configurations (states) in which the program can stay indefinitely (e.g. [12, 25, 37]). For instance, the work of [30] considers computing "lassos" (where a lasso is a finite prefix followed by a finite cycle infinitely repeated) as counter-examples for termination and presents a trace-based approach based

on lassos to prove non-termination of deterministic programs. In general, finite lassos are not sufficient to witness non-termination. While lassos are periodic, proving non-termination for programs with aperiodic infinite traces via set-based methods has been considered in [12, 37] for programs with non-determinism. In [12], a method is proposed where "closed recurrence sets" of configurations are used to prove non-termination. Intuitively, a closed recurrence set must contain some initial configuration, must contain no terminal configurations, and cannot be escaped once entered. In [12], closed recurrence sets are defined with respect to under-approximations of the transition relation, and an under-approximation search guided by several calls to a safety prover is used to compute a closed recurrence set. In [37], a constraint solving-based method is proposed to search for "quasi-invariants" (sets of configurations which cannot be left once entered) exhaustively in all strongly-connected subgraphs. A safety prover is used to check reachability for every obtained quasi-invariant. For constraint solving, Max-SMT is used in [37].

Limitations of previous approaches. While the previous works represent significant advancement for proving non-termination, each of them has, to our best knowledge, at least one of the following limitations:

a) They do not support non-determinism, e.g. [29, 46].
b) They only work for lassos (i.e. periodic non-terminating traces), e.g. [30].
c) *Theoretical limitation* of not providing any *(relative) completeness guarantees.* Clearly, a non-termination proving algorithm cannot be both sound and complete, since non-termination is well-known to be undecidable. However, as in the case of termination proving, it can be beneficial to provide relative completeness guarantees, i.e. conditions on the input program under which the algorithm is guaranteed to prove non-termination. To our best knowledge, the only approaches with such guarantees are [29, 39]; however, both of them only provide guarantees for a certain class of *deterministic* programs.
d) Most of the previous approaches do not support programs with polynomial arithmetic (with an exception of [18, 24]).

Our contributions. In this work we propose a new set-based approach to non-termination proving in integer programs. Intuitively, it searches for a diverging program configuration, i.e. a configuration that is reachable but from which no program run is terminating (after resolving non-determinism using symbolic polynomial assignments). Our approach is based on a simple technique of *program reversal*, which reverses each transition in the program's transition system to produce the reversed transition system. The key property of this construction is that, given a program configuration, there is a terminating run starting in it if and only if it is reachable from the terminal location in the reversed transition system. This allows over-approximating the set of all program configurations from which termination can be reached by computing an invariant in the program's reversed transition system. We refer to the invariants in reversed transition systems as *backward invariants*. To generate the backward invariant, we may employ state-of-the-art polynomial invariant generation techniques to the reversed transition system as a single-shot procedure which is the main practical benefit of the program reversal. Our method proves non-termination by generating a backward invariant whose complement is reachable. Hence, our new method adapts the classical and well-studied techniques for inductive invariant generation in order to find non-termination proofs by combining forward and backward analysis of a program. While such a combined analysis is common in safety analysis where the goal is to show that no program run reaches some annotated set of configurations [6], to our best knowledge it has never been considered for proving non-termination in programs with non-determinism, where we need to find a single program run that does not terminate. The key features of our method are as follows:

a) Our approach supports programs with non-determinism.
b) Our approach is also applicable to programs where all non-terminating traces are aperiodic.
c) *Relative completeness guarantee:* The work of [12] establishes that closed recurrence sets with respect to under-approximations are a sound and complete *certificate* of non-termination, yet the algorithm based on these certificates does not in itself provide any relative completeness guarantee (in the above sense). For our approach we show the following: If there is an under-approximation of the transition relation where non-determinism can be resolved by polynomial assignments such that the resolved program contains a closed recurrence set representable as a propositional predicate map, then our approach is guaranteed to prove non-termination. We obtain such guarantee by employing relatively complete methods for inductive invariant synthesis, which is another key advantage of adapting invariant generation techniques to non-termination proving. Moreover, we provide even stronger relative completeness guarantees for programs in which non-determinism appears only in branching (but not in variable assignments).
d) Our approach supports programs with polynomial arithmetic.

We developed a prototype tool RevTerm which implements our approach. We experimentally compared our tool with state-of-the-art non-termination provers on standard benchmarks from the Termination and Complexity Competition (TermComp'19 [26]). Our tool demonstrates performance on par with the most efficient of the competing provers, while providing additional guarantees. In particular, with a proper configuration, our tool achieved the largest number of benchmarks proved non-terminating.

Outline. After presenting the necessary definitions (Section 2), we present our approach and its novel aspects in the following order: first we introduce the technique of program reversal (Section 3); then we present a new certificate for non-termination (Section 4) based on so-called *backward invariants,* as well as an invariant generation-based automated approach for this certificate (Section 5); finally we prove relative completeness guarantees (Section 5.4). We conclude with the presentation of our experiments and discussion of related work.

2 Preliminaries

Syntax of programs. In this work we consider simple imperative arithmetic programs with polynomial integer arithmetic and with non-determinism. They consist of standard programming constructs such as conditional branching, while-loops and (deterministic) variable assignments. In addition, we allow constructs for non-deterministic assignments of the form $x := \textbf{ndet}()$, which assign any integral value to x. The adjective *polynomial* refers to the fact that all arithmetic expressions are polynomials in program variables.

Example 2.1 (Running example)**.** Fig. 1 left shows a program which will serve as our running example. The second line contains a non-deterministic assignment, in which any integral value can be assigned to the variable x.

Removing non-deterministic branching. We may *without loss of generality* assume that non-determinism does not appear in branching: for the purpose of termination analysis, one can replace each non-deterministic branching with a non-deterministic assignment. Indeed, non-deterministic branching in programs is given by a command **if** $*$ **then**, meaning that the control-flow can follow any of the two subsequent branches. By introducing an auxiliary program variable $x_{\textbf{ndet}}$ and replacing each command **if** $*$ **then** with two commands

$$x_{\textbf{ndet}} := \textbf{ndet}()$$
$$\textbf{if } x_{\textbf{ndet}} \geq 0 \textbf{ then}$$

we obtain a program which terminates on every input if and only if the original program does. This removal is done for the sake of easier presentation and neater definition of the *resolution of non-determinism*, see Section 5.1.

Predicate, assertion, propositional predicate. We use the following terminology:

- *Predicate*, which is a set of program variable valuations.
- *Assertion*, which is a finite conjunction of polynomial inequalities over program variables. We need not differentiate between non-strict and strict inequalities since we work over integer arithmetic.
- *Propositional predicate (PP)*, which is a finite disjunction of assertions.

We write $\mathbf{x} \models \phi$ to denote that the predicate ϕ given by a formula over program variables is satisfied by substituting values in $\mathbf{x}$ for corresponding variables in ϕ. For a predicate ϕ, we define $\neg\phi = \mathbb{Z}^{|\mathcal{V}|} \setminus \phi$.

Transition system. We model programs using transition systems [15].

Definition 2.2 (Transition system)**.** *A transition system is a tuple* $\mathcal{T} = (L, \mathcal{V}, \ell_{init}, \Theta_{init}, \mapsto)$*, where* L *is a finite set of* locations*;* $\mathcal{V}$ *is a finite set of* program variables*;* ℓ_{init} *is the* initial location*;* Θ_{init} *is the set of* initial variable valuations*; and* $\mapsto \subseteq L \times L \times \mathcal{P}(\mathbb{Z}^{|\mathcal{V}|} \times \mathbb{Z}^{|\mathcal{V}|})$ *is a finite set of* transitions*. Each transition is defined as an ordered triple* $\tau = (l, l', \rho_\tau)$*, with* l *its* source *and* l' *the* target location*, and the* transition relation $\rho_\tau \subseteq \mathbb{Z}^{|\mathcal{V}|} \times \mathbb{Z}^{|\mathcal{V}|}$*. The transition relation is usually given by an assertion over* $\mathcal{V}$ *and* $\mathcal{V}'$*, where* $\mathcal{V}$ *represents the source-state variables and* $\mathcal{V}'$ *the target-state variables.*

Each program P naturally defines a transition system $\mathcal{T}$, with each transition relation given by an assertion over program variables. Its construction is standard and we omit it. The only difference is that here ℓ_{init} will correspond to the first non-assignment command in the program code, whereas the sequence of assignments preceding ℓ_{init} specifies Θ_{init} (unspecified variables may take any value). Hence Θ_{init} will also be an assertion. For transition systems derived from programs, we assume the existence of a special *terminal location* ℓ_{out}, which represents a "final" line of the program code. It has a single outgoing transition which is a self-loop with a transition relation $\rho = \{(\mathbf{x}, \mathbf{x}) \mid \mathbf{x} \in \mathbb{Z}^{|\mathcal{V}|}\}$.
A *configuration* (or *state*) of a transition system $\mathcal{T}$ is an ordered pair $(l, \mathbf{x})$ where l is a location and $\mathbf{x}$ is a vector of variable valuations. A configuration $(l', \mathbf{x}')$ is a *successor* of a configuration $(l, \mathbf{x})$ if there is a transition $\tau = (l, l', \rho_\tau)$ with $(\mathbf{x}, \mathbf{x}') \in \rho_\tau$. The self-loop at ℓ_{out} allows us to without loss of generality assume that each configuration has at least one successor in a transition system $\mathcal{T}$ derived from a program. Given a configuration $\mathbf{c}$, a *finite path from* $\mathbf{c}$ in $\mathcal{T}$ is a finite sequence of configurations $\mathbf{c} = (l_0, \mathbf{x}_0), \ldots, (l_k, \mathbf{x}_k)$ where for each $0 \leq i < k$ we have that $(l_{i+1}, \mathbf{x}_{i+1})$ is a successor of $(l_i, \mathbf{x}_i)$. A *run* (or *execution*) *from* $\mathbf{c}$ in $\mathcal{T}$ is an infinite sequence of configurations whose every finite prefix is a finite path from $\mathbf{c}$. A configuration is said to be *initial* if it belongs to the set $\{(\ell_{init}, \mathbf{x}) \mid \mathbf{x} \models \Theta_{init}\}$. A configuration $(l, \mathbf{x})$ is *reachable from* $\mathbf{c}$ if there is a finite path from $\mathbf{c}$ with the last configuration $(l, \mathbf{x})$. When we omit specifying the configuration $\mathbf{c}$, we refer to a finite path, execution and reachability from some initial configuration. A configuration $(l, \mathbf{x})$ is said to be *terminal* if $l = \ell_{out}$.

Example 2.3. The transition system for our running example is presented in Fig. 1 center. It contains 6 locations $L = \{l_0, l_1, l_2, l_3, l_4, \ell_{out}\}$ with $\ell_{init} = l_0$, and two program variables $\mathcal{V} = \{x, y\}$. Since there are no assignments preceding the initial program location, we have $\Theta_{init} = \mathbb{Z}^2$. Locations are depicted by labeled circles, transitions by directed arrows

between program locations and their transition relations are given in the associated rectangular boxes.

Invariants and inductive predicate maps. Given a transition system $\mathcal{T}$, a *predicate map* is a map I assigning to each location in $\mathcal{T}$ a predicate over the program variables. A predicate map naturally defines a set of configurations in $\mathcal{T}$ and we will freely interchange between the two notions. A predicate map is *of type-*(c, d) if it assigns to each program location a propositional predicate which is a disjunction of d assertions, each being a conjunction of c polynomial inequalities. For a predicate map I, we define the complement predicate map $\neg I$ as $(\neg I)(l) = \neg I(l)$ for each location l.
A predicate map I is said to be an *invariant* if for every reachable configuration $(l, \mathbf{x})$ in $\mathcal{T}$, we have $\mathbf{x} \models I(l)$. Intuitively, invariants are over-approximations of the set of reachable configurations in the transition system. A predicate map is *inductive* if it is inductive with respect to every transition $\tau = (l, l', \rho_\tau)$, i.e. if for any pair of configurations $(l, \mathbf{x})$ and $(l', \mathbf{x}')$ with $\mathbf{x} \models I(l)$ and $(\mathbf{x}, \mathbf{x}') \in \rho_\tau$, we also have $\mathbf{x}' \models I(l')$.

Termination problem. Given a program and its transition system $\mathcal{T}$, we say that a run reaching ℓ_{out} is *terminating*. The program is said to be *terminating* if every run in $\mathcal{T}$ is terminating. Otherwise it is said to be *non-terminating*. One witness to non-termination can be a configuration that is reachable but from which there are no terminating executions. We call such configuration *diverging*.

Example 2.4. Consider again the running example in Fig. 1. For any initial configuration with $x \geq 9$, executions that always assign $x := 9$ when passing the non-deterministic assignment are non-terminating. On the other hand, the execution that assigns $x := 0$ in the non-deterministic assignment enters the outer loop only once and then terminates. Thus, no initial configuration is diverging. One can similarly check that other configurations are also not diverging.

3 Transition System Reversal

We now show that it is possible to "reverse" a transition system by reversing each of its transitions. This construction is the core concept of our approach to proving non-termination, since configurations in the program from which ℓ_{out} is reachable will be precisely those configurations which can be reached from ℓ_{out} in the reversed transition system. We then present a sound and complete certificate for non-termination based on this construction.

Definition 3.1 (Reversed transition system)**.** Given a transition system $\mathcal{T} = (L, \mathcal{V}, \ell_{init}, \Theta_{init}, \mapsto)$ and a transition $\tau = (l, l', \rho_\tau) \in \mapsto$, let

$$\rho'_\tau = \{(\mathbf{x}', \mathbf{x}) \mid (\mathbf{x}, \mathbf{x}') \in \rho_\tau\}.$$

If ρ_τ is given by an assertion over $\mathcal{V} \cup \mathcal{V}'$, ρ'_τ is obtained from ρ_τ by replacing each unprimed variable in the defining assertion for ρ_τ with its primed counterpart, and vice-versa. Then for an assertion Θ, we define the *reversed transition system* of $\mathcal{T}$ with initial variable valuations Θ as a tuple $\mathcal{T}^{r,\Theta} = (L, \mathcal{V}, \ell_{out}, \Theta, \mapsto^r)$, where $\mapsto^r = \{(l', l, \rho'_\tau) \mid (l, l', \rho_\tau) \in \mapsto\}$.

Note that this construction satisfies Definition 2.2 and thus yields another transition system. All notions that were defined before (e.g. configuration, finite path, etc.) are defined analogously for the reversed transition systems.

Example 3.2. Fig. 1 right shows the reversed transition system $\mathcal{T}^{r,\Theta}$ of the program in Fig. 1. Note that for every transition τ in $\mathcal{T}$ for which ρ_τ is given by a conjunction of an assertion over unprimed program variables and $x' = x \wedge y' = y$, after reversing we obtain the conjunction of the same assertion just now over primed variables and $x' = x \wedge y' = y$. Hence, for such τ the transition relation is invariant under reversing. For example, a transition from l_0 to l_1 in $\mathcal{T}$ has transition relation $x \geq 9 \wedge x' = x \wedge y' = y$ so the reversed transition has transition relation $x' \geq 9 \wedge x = x' \wedge y = y'$. As $x' = x$, this is the same relation as prior to reversal.

The following lemma is the key property of this construction.

Lemma 3.3 (Key property of reversed transition systems)**.** *Let $\mathcal{T}$ be a transition system, Θ an assertion and $\mathcal{T}^{r,\Theta}$ the reversed transition system of $\mathcal{T}$ with initial variable valuations Θ. Let $\mathbf{c}$ and $\mathbf{c}'$ be two configurations. Then $\mathbf{c}'$ is reachable from $\mathbf{c}$ in $\mathcal{T}$ if and only if $\mathbf{c}$ is reachable from $\mathbf{c}'$ in $\mathcal{T}^{r,\Theta}$.*

Proof. We prove that if $\mathbf{c}'$ is reachable from $\mathbf{c}$ in $\mathcal{T}$ then $\mathbf{c}$ is reachable from $\mathbf{c}'$ in $\mathcal{T}^{r,\Theta}$, the other direction follows analogously. Suppose that $\mathbf{c} = (l_0, \mathbf{x}_0), (l_1, \mathbf{x}_1), \ldots, (l_k, \mathbf{x}_k) = \mathbf{c}'$ is a path from $\mathbf{c}$ to $\mathbf{c}'$ in $\mathcal{T}$. Then for each $0 \leq i < k$ there is a transition $\tau_i = (l_i, l_{i+1}, \rho_{\tau_i})$ in $\mathcal{T}$ for which $(\mathbf{x}_i, \mathbf{x}_{i+1}) \in \rho_{\tau_i}$. But then $(\mathbf{x}_{i+1}, \mathbf{x}_i) \in \rho'_{\tau_i}$ and $\tau^r_i = (l', l, \rho'_{\tau_i})$, hence $(l_i, \mathbf{x}_i)$ is a successor of $(l_{i+1}, \mathbf{x}_{i+1})$ in $\mathcal{T}^{r,\Theta}$. Thus $\mathbf{c}' = (l_k, \mathbf{x}_k), (l_{k-1}, \mathbf{x}_{k-1}), \ldots, (l_0, \mathbf{x}_0) = \mathbf{c}$ is a finite path in $\mathcal{T}^{r,\Theta}$, proving the claim. □

Backward Invariants. Lemma 3.3 implies that generating invariants for the reversed transition system $\mathcal{T}^{r,\Theta}$ provides a way to over-approximate the set of configurations in $\mathcal{T}$ from which some configuration in the set $\{(\ell_{out}, \mathbf{x}) \mid \mathbf{x} \models \Theta\}$ is reachable. This motivates the notion of a *backward invariant*, which will be important in what follows.

Definition 3.4 (Backward invariant)**.** For a transition system $\mathcal{T}$ and an assertion Θ, we say that the predicate map BI is a *backward invariant in* $\mathcal{T}^{r,\Theta}$ if it is an invariant in $\mathcal{T}^{r,\Theta}$. The word backward is used to emphasize that we are working in the reversed transition system.

We conclude this section with a theorem illustrating the behavior of inductive predicate maps under program reversal.

Theorem 3.5. *Let $\mathcal{T}$ be a transition system, Θ an assertion, I a predicate map and $\mathcal{T}^{r,\Theta}$ the reversed transition system. Then I is inductive in $\mathcal{T}$ if and only if $\neg I$ is inductive in $\mathcal{T}^{r,\Theta}$.*

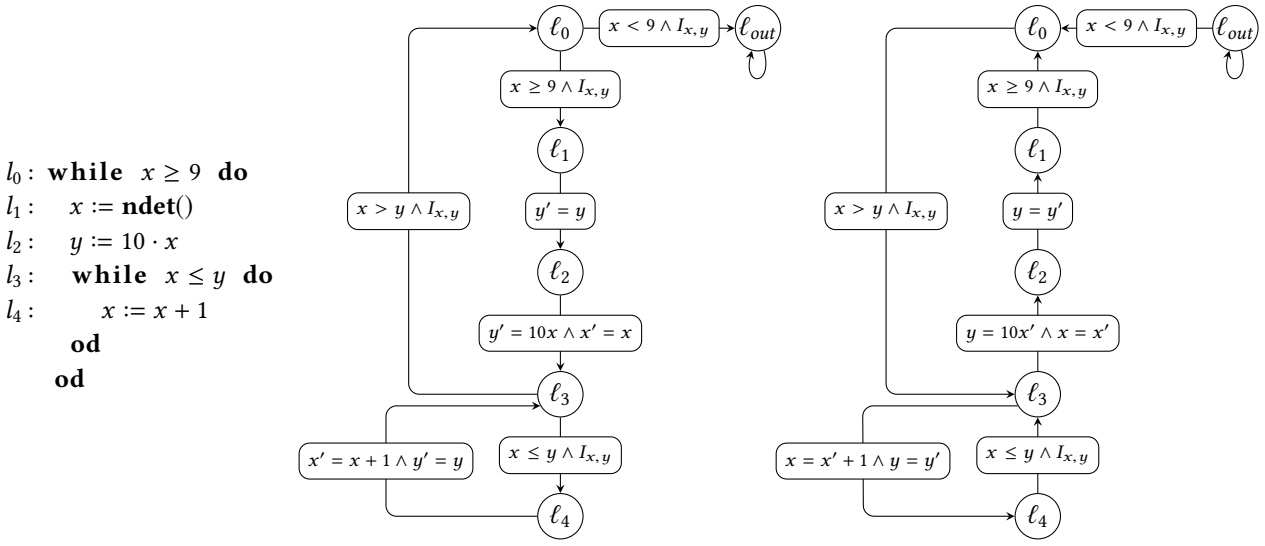

Figure 1. Running example, its associated transition system, and its reversed transition system. $I_{x,y}$ denotes $x' = x \wedge y' = y$ and is used for readability.

Proof. We show that I being inductive in $\mathcal{T}$ implies that $\neg I$ is inductive in $\mathcal{T}^{r,\Theta}$. The other direction of the lemma follows analogously.
Let $\tau^r = (l', l, \rho'_\tau)$ be a transition in $\mathcal{T}^{r,\Theta}$ obtained by reversing $\tau = (l, l', \rho_\tau)$ in $\mathcal{T}$. Assume that $\mathbf{x}' \in \neg I(l')$. To show inductiveness of $\neg I$ in the reversed transition system, we take a successor $(l, \mathbf{x})$ of $(l', \mathbf{x}')$ in the reversed transition system with $(\mathbf{x}', \mathbf{x}) \in \rho'_\tau$, and we need to show that $\mathbf{x} \in \neg I(l)$. By definition of the reversed transition we have $(\mathbf{x}, \mathbf{x}') \in \rho_\tau$. So, if on the contrary we had $\mathbf{x} \in I(l)$, inductiveness of I in $\mathcal{T}$ would imply that $\mathbf{x}' \in I(l')$. This would contradict the assumption that $\mathbf{x}' \in \neg I(l')$. Thus, we must have $\mathbf{x} \in \neg I(l)$, and $\neg I$ is inductive in $\mathcal{T}^{r,\Theta}$. □

4 Sound and Complete Certificate for Non-termination

Lemma 3.3 indicates that reversed transition systems are relevant for the termination problem, as they provide means to describe configurations from which the terminal location can be reached. We now introduce the *BI-certificate for non-termination*, based on the reversed transition systems and backward invariants. We show that it is both *sound* and *complete* for proving non-termination and hence characterizes it (i.e. a program is non-terminating if and only if it admits the certificate). This is done by establishing a connection to recurrence sets [12, 30], a notion which provides a necessary and sufficient condition for a program to be non-terminating.

Recurrence set. A *recurrence set* [30] in a transition system $\mathcal{T}$ is a non-empty set of configurations $\mathcal{G}$ which (1) contains some configuration reachable in $\mathcal{T}$, (2) every configuration in $\mathcal{G}$ has at least one successor in $\mathcal{G}$, and (3) contains no terminal configurations. The last condition was not present in [30] and we add it to account for the terminal location and the self-loop at it, but the definitions are easily seen to be equivalent. In [30], it is shown that a program is non-terminating if and only if its transition system contains a recurrence set. The work in [12] notes that one may without loss of generality restrict attention to recurrence sets which contain some initial configuration (which they call *open recurrence sets*). Indeed, to every recurrence set one can add configurations from some finite path reaching it to obtain an open recurrence set, and there is at least one such path since each recurrence set contains a reachable configuration.

Closed recurrence set. A *closed recurrence set* [12] is an open recurrence set C with the additional property of being inductive, i.e. for every configuration in C *each* of its successors is also contained in C. The work [12, Theorems 1 and 2] shows that closed recurrence sets can be used to define a sound and complete certificate for non-termination, which we describe next. Call $U = (L, \mathcal{V}, \ell_{init}, \Theta_{init}, \mapsto_U)$ an *under-approximation* of $\mathcal{T} = (L, \mathcal{V}, \ell_{init}, \Theta_{init}, \mapsto)$ if for every $(l, l', \rho^u_\tau) \in \mapsto_U$ there exists $(l, l', \rho_\tau) \in \mapsto$ with $\rho^u_\tau \subseteq \rho_\tau$. Then $\mathcal{T}$ contains an open recurrence set if and only if there is an under-approximation U of $\mathcal{T}$ and a closed recurrence set in U.

Proper under-approximations. We introduce a notion of proper under-approximation. An under-approximation U of $\mathcal{T}$ is *proper* if every configuration which has a successor in $\mathcal{T}$ also has at least one successor in U. This is a new concept and restricts general under-approximations, but it will be relevant in defining the *BI*-certificate for non-termination and establishing its soundness and completeness. The next lemma is technical and shows that closed recurrence sets in proper under-approximations are sound and complete

for proving non-termination, its proof can be found in the extended version of the paper [11].

Lemma 4.1. *Let P be a non-terminating program and $\mathcal{T}$ its transition system. Then there exist a proper under-approximation U of $\mathcal{T}$ and a closed recurrence set C in U.*

BI-certificate for non-termination. We introduce and explain how backward invariants in combination with proper under-approximations can be used to characterize non-termination. Suppose P is a program we want to show is non-terminating, and $\mathcal{T}$ is its transition system. Let $Reach_{\mathcal{T}}(\ell_{out})$ be the set of variable valuations of all reachable terminal configurations in $\mathcal{T}$. A *BI-certificate for non-termination* will consist of an ordered triple (U, BI, Θ) of a proper under-approximation U of $\mathcal{T}$, a predicate map BI and an assertion Θ such that

- $\Theta \supseteq Reach_{\mathcal{T}}(\ell_{out})$;
- BI is an inductive backward invariant in $U^{r,\Theta}$;
- BI is not an invariant in $\mathcal{T}$.

Theorem 4.2 (Soundness of our certificate)**.** *Let P be a program and $\mathcal{T}$ its transition system. If there exists a BI-certificate (U, BI, Θ) in $\mathcal{T}$, then P is non-terminating.*

Proof sketch. As BI is not an invariant in $\mathcal{T}$, its complement $\neg BI$ contains a reachable configuration $\mathbf{c}$. On the other hand, BI is inductive in $U^{r,\Theta}$ so by Theorem 3.5 $\neg BI$ is inductive in U. Since U is proper (and since in transition systems induced by programs every configuration has a successor), one may take a finite path reaching $\mathbf{c}$ and inductively keep picking successors in U from $\mathbf{c}$, obtaining an execution whose all but finitely many configurations are in $\neg BI$. By the definition of Θ and since BI is an invariant for $U^{r,\Theta}$, $\neg BI$ contains no reachable terminal configuration hence this execution is non-terminating. Details can be found in the extended version of the paper [11]. □

Example 4.3. Consider again the running example and its transition system $\mathcal{T}$ presented in Fig. 1. Let U be the under-approximation of $\mathcal{T}$ defined by restricting the transition relation of the non-deterministic assignment $x := \mathbf{ndet}()$ as $\rho_\tau^U = \{(x, y, x', y') \mid x' = 9, y' = y\}$. Intuitively, U is a transition system of the program obtained by replacing the non-deterministic assignment in P with $x := 9$. Define a predicate map BI as

$$BI(l) = \begin{cases} (1 \geq 0) & \text{if } l = \ell_{out} \\ (x \leq 8) & \text{if } l \in \{l_0, l_2, l_3, l_4\} \\ (-1 \geq 0) & \text{if } l = l_1, \end{cases}$$

i.e. $BI(l_1)$ is empty, and let $\Theta = \mathbb{Z}^2$. $U^{r,\Theta}$ can be obtained from $\mathcal{T}^{r,\Theta}$ by replacing the transition relation from l_2 to l_1 with $x = 9 \wedge y = y'$ in Fig. 1 right. Then U is proper, and BI is an inductive backward invariant for $U^{r,\Theta}$ since no transition can increase x. On the other hand, $(l_0, 9, 0)$ is reachable in $\mathcal{T}$ but not contained in BI, thus BI is not an invariant in $\mathcal{T}$. Hence (U, BI, Θ) is a BI-certificate for non-termination and the program is non-terminating.

By making a connection to closed recurrence sets, the following theorem shows that backward invariants in combination with proper under-approximations of $\mathcal{T}$ also provide a *complete characterization* of non-termination.

Theorem 4.4 (Complete characterization of non-termination)**.** *Let P be a non-terminating program with transition system $\mathcal{T}$. Then $\mathcal{T}$ admits a proper under-approximation U and a predicate map BI such that BI is an inductive backward invariant in the reversed transition system $U^{r,\mathbb{Z}^{|V|}}$, but not an invariant in $\mathcal{T}$.*

Proof sketch. Since P is non-terminating, from Lemma 4.1 we know that $\mathcal{T}$ admits a proper under-approximation U and a closed recurrence set C in U. For each location l in $\mathcal{T}$, let $C(l) = \{\mathbf{x} \mid (l, \mathbf{x}) \in C\}$. Define the predicate map BI as $BI(l) = \neg C(l)$ for each l. Then, using Theorem 3.5 one can show that U and BI satisfy the conditions of the theorem. For details, see the extended version of the paper [11]. □

Remark 1 (Connection to the *pre*-operator). There is a certain similarity between reversal of an individual transition and application of the *pre*-operator, the latter being a well known concept in program analysis. However, in our approach we introduce reversed transition systems which are obtained by reversing *all transitions* (hence the name "program reversal"). This allows us using black-box invariant generation techniques as a *one-shot* method of computing sets from which a terminal location can be reached, as presented in the next section. This is in contrast to approaches which rely on an iterative application of the *pre*-operator.

5 Algorithm for Proving Non-termination

We now present our algorithm for proving non-termination based on program reversing and BI-certificates introduced in Section 4. It uses a black box constraint solving-based method for generating (possibly disjunctive) inductive invariants, as in [10, 15, 29, 34–36, 42, 43]. This is a classical approach to invariant generation and it fixes a template for the invariant (i.e. a type-(c, d) propositional predicate map as well as an upper bound D on the degree of polynomials, where c, d and D are provided by the user), introduces a fresh variable for each template coefficient, and encodes invariance and inductiveness conditions as existentially quantified constraints on template coefficient variables. The obtained system is then solved and any solution yields an inductive invariant. Moreover, the method is relatively complete [10, 42, 43] in the sense that every inductive invariant of the fixed template and maximal polynomial degree is a solution to the system of constraints. Efficient practical approaches to polynomial inductive invariant generation have been presented in [35, 36].

We first introduce *resolution of non-determinism* which induces a type of proper under-approximations of the program's transition system of the form that allows searching for them via constraint solving. We then proceed to our main algorithm. In what follows, P will denote a program with polynomial arithmetic and $\mathcal{T} = (L, \mathcal{V}, \ell_{init}, \Theta_{init}, \mapsto)$ will be its transition system.

5.1 Resolution of Non-determinism

As we saw in Example 2.4, there may exist non-diverging program configurations which become diverging when supports of non-deterministic assignments are restricted to suitably chosen subsets. Here we define one such class of restrictions which "resolves" each non-deterministic assignment by replacing it with a polynomial expression over program variables. Such resolution ensures that the resulting under-approximation of the program's transition relation is proper. Let $T_{NA} \subseteq \mapsto$ be the set of transitions corresponding to non-deterministic assignments in P.

Definition 5.1 (Resolution of non-determinism)**.** A *resolution of non-determinism* for $\mathcal{T}$ is a map R^{NA} which to each $\tau \in T_{NA}$ assigns a polynomial expression $R^{NA}(\tau)$ over program variables. It naturally defines a *restricted transition system* $\mathcal{T}_{R^{NA}}$ which is obtained from $\mathcal{T}$ by letting the transition relation of $\tau \in T_{NA}$ corresponding to an assignment $x := \textbf{ndet}()$ be

$$\rho_\tau^{R^{NA}}(\mathbf{x}, \mathbf{x}') := (x' = R^{NA}(\tau)(\mathbf{x})) \wedge \bigwedge_{y \in \mathcal{V} \setminus \{x\}} y' = y.$$

Note that $\mathcal{T}_{R^{NA}}$ is a proper under-approximation of $\mathcal{T}$. If there exists a resolution of non-determinism R^{NA} and a configuration $\mathbf{c}$ which is reachable in $\mathcal{T}$ but from which no execution in $\mathcal{T}_{R^{NA}}$ terminates, then any such execution is non-terminating in $\mathcal{T}$ as well. We say that any such configuration $\mathbf{c}$ is *diverging with respect to (w.r.t.)* R^{NA}.

Example 5.2. Looking back at the program in Figure 1, define a resolution of non-determinism R^{NA} to assign constant expression 9 to the non-deterministic assignment $x := \textbf{ndet}()$. Then every initial configuration with $x \geq 9$ becomes diverging w.r.t. R^{NA}.

5.2 Algorithm

Main idea. To prove non-termination, our algorithm uses a constraint solving approach to find a *BI*-certificate. It searches for a resolution of non-determinism R^{NA}, a propositional predicate map *BI* and an assertion Θ such that:

1. $\Theta \supseteq \mathit{Reach}_{\mathcal{T}}(\ell_{out})$ (recall that $\mathit{Reach}_{\mathcal{T}}(\ell_{out})$ is the set of variable valuations of all reachable terminal configurations in $\mathcal{T}$);
2. *BI* is an inductive backward invariant for the reversed transition system $\mathcal{T}^{r,\Theta}_{R^{NA}}$;
3. *BI* is not an invariant for $\mathcal{T}$.

Need for inductive invariants and safety checking. Using the aforementioned black box invariant generation, our algorithm encodes the conditions on R^{NA}, *BI*, and Θ as polynomial constraints and then solves them. However, the method is only able to generate *inductive* invariants, which is to say that encoding "*BI* is not an invariant for $\mathcal{T}$ " is not possible. Instead, we modify the third requirement on *BI* above to get:

1. $\Theta \supseteq \mathit{Reach}_{\mathcal{T}}(\ell_{out})$;
2. *BI* is an inductive backward invariant for $\mathcal{T}^{r,\Theta}_{R^{NA}}$;
3. *BI* is not an inductive invariant for $\mathcal{T}$.

The third requirement does not guarantee that we get a proper *BI*-certificate. However it guides invariant generation to search for *BI* which is less likely to be an invariant for $\mathcal{T}$. It follows that the algorithm needs to do additional work to ensure that the triple (R^{NA}, BI, Θ) is a *BI*-certificate.

Splitting the algorithm into two checks. The predicate map *BI* is not an inductive invariant for $\mathcal{T}$ if and only if it has one of the following properties: either it does not contain some initial configuration or is not inductive with respect to some transition in $\mathcal{T}$. For each of these two properties, we can separately compute *BI* satisfying it and the properties (1) and (2) above, followed by a check whether the computed *BI* indeed proves non-termination. We refer to these two independent computations as two checks of our algorithm:

- *Check 1* - the algorithm checks if there exist R^{NA}, *BI* and Θ as above so that *BI* does not contain some initial configuration and conditions (1) and (2) are satisfied. By Theorem 3.5, *BI* is inductive for $\mathcal{T}^{r,\Theta}_{R^{NA}}$ if and only if the complement $\neg BI$ is an inductive predicate map for $\mathcal{T}_{R^{NA}}$. Moreover, since $\neg BI$ contains an initial configuration there is no need for an additional reachability check to conclude that *BI* is not an invariant for $\mathcal{T}$. Hence by fixing $\Theta = \mathbb{Z}^{|\mathcal{V}|}$, to prove non-termination it suffices to check if there exist a resolution of non-determinism R^{NA}, a predicate map I and an initial configuration $\mathbf{c}$ in $\mathcal{T}$ such that I contains $\mathbf{c}$, I is inductive for $\mathcal{T}_{R^{NA}}$ and $I(\ell_{out}) = \emptyset$.
- *Check 2* - the algorithm checks if there exist R^{NA}, Θ and *BI* as above so that *BI* is not inductive in $\mathcal{T}$ and conditions (1) and (2) are satisfied. If a solution is found, the algorithm still needs to find a configuration in $\neg BI$ which is reachable in $\mathcal{T}$, via a call to a safety prover.

Algorithm summary. As noted at the beginning of Section 5, the invariant generation method first needs to fix a template for the propositional predicate map and the maximal polynomial degree. Thus our algorithm is parametrized by c and d which are bounds on the template size of propositional predicate maps (d being the maximal number of disjunctive clauses and c being the maximal number of conjunctions in each clause), and by an upper bound D on polynomial degrees. The algorithm consists of two checks, which can be executed either sequentially or in parallel:

Algorithm 1: Proving non-termination

input : A program P, its transition system $\mathcal{T}$, predicate map template size (c, d), maximal polynomial degree D.

output : Proof of non-termination if found, otherwise "Unknown"

set a template for each polynomial defined by resolution of non-determinism R^{NA}
construct restricted transition system $\mathcal{T}_{R^{NA}}$
set templates for configuration $\mathbf{c}$ and for an invariant I of type-(c, d)
encode $\Phi_1 = \phi_{\mathbf{c}} \wedge \phi_{I, R^{NA}}$
if Φ_1 *feasible* **then return** Non-termination
else
  set templates for invariant $\tilde{I}$ of type-$(c, 1)$ and for a backward invariant BI of type-(c, d)
  construct reversed transition system $\mathcal{T}^{r, \tilde{I}(\ell_{out})}_{R^{NA}}$
  foreach $\tau \in \mapsto$ **do** set templates for $\mathbf{x}_\tau$, $\mathbf{x}'_\tau$
  encode $\Phi_2 = \phi_{\tilde{I}} \wedge \phi_{BI, R^{NA}} \wedge \bigvee_{\tau \in \mapsto} \phi_\tau$
  if Φ_2 *feasible* **then**
    if $\exists\, (l, \mathbf{x})$ *Reachable in* $\mathcal{T}$ *with* $\mathbf{x} \models \neg BI(l)$ **then return** Non-termination
    else return Unknown
  else return Unknown

Check 1 - the algorithm checks if there exist a resolution of non-determinism R^{NA}, a predicate map I and an initial configuration $\mathbf{c}$ such that (1) I is an inductive invariant in $\mathcal{T}_{R^{NA}}$ for the single initial configuration $\mathbf{c}$, and (2) $I(\ell_{out}) = \emptyset$. To do this, we fix a template for each of R^{NA}, I and $\mathbf{c}$, and encode these properties as polynomial constraints:

- For each transition τ in T_{NA}, fix a template for a polynomial $R^{NA}(\tau)$ over program variables of degree at most D. That is, introduce a fresh template variable for each coefficient of such a polynomial.
- Introduce fresh variables $c_1, c_2, \ldots, c_{|\mathcal{V}|}$ defining the variable valuation of $\mathbf{c}$. Then substitute these variables into the assertion Θ_{init} specifying initial configurations in $\mathcal{T}$ to obtain the constraint $\phi_{\mathbf{c}}$ for $\mathbf{c}$ being an initial configuration.
- Fix a template for the propositional predicate map I of type-(c, d) and maximal polynomial degree D. The fact that I is an inductive invariant for $\mathcal{T}_{R^{NA}}$ with the single initial configuration $\mathbf{c}$ and $I(\ell_{out}) = \emptyset$ is encoded by the invariant generation method (e.g. [15, 42]) into a constraint $\phi_{I, R^{NA}}$.

The algorithm then tries to solve $\Phi_1 = \phi_{\mathbf{c}} \wedge \phi_{I, R^{NA}}$ using an off-the-shelf SMT solver. If a solution is found, $\mathbf{c}$ is an initial diverging configuration w.r.t. $\mathcal{T}_{R^{NA}}$, so the algorithm reports non-termination.

Check 2 - the algorithm checks if there exist a resolution of non-determinism R^{NA}, an assertion Θ, a predicate map BI and a transition $\tau \in T_{NA}$ such that (1) $\Theta \supseteq \mathit{Reach}_{\mathcal{T}}(\ell_{out})$, (2) BI is an inductive backward invariant for $\mathcal{T}^{r, \Theta}_{R^{NA}}$, and (3) BI is not inductive w.r.t. τ in $\mathcal{T}$. To encode $\Theta \supseteq \mathit{Reach}_{\mathcal{T}}(\ell_{out})$, we introduce another propositional predicate map $\tilde{I}$ (purely conjunctive for the sake of efficiency), and impose a requirement on it to be an inductive invariant for $\mathcal{T}$. We may then define the initial variable valuations for $\mathcal{T}^{r, \Theta}_{R^{NA}}$ as $\Theta = \tilde{I}(\ell_{out})$. The algorithm introduces fresh template variables for R^{NA}, $\tilde{I}$ and BI, as well as for a pair of variable valuations $\mathbf{x}_\tau$ and $\mathbf{x}'_\tau$ for each transition $\tau = (l, l', \rho_\tau)$ in $\mathcal{T}$ and imposes the following constraints:

- For each transition τ in T_{NA}, fix a template for a polynomial expression $R^{NA}(\tau)$ of degree at most D over program variables.
- Fix a template for the propositional predicate map $\tilde{I}$ of type-$(c, 1)$ (as explained above, for efficiency reasons we make $\tilde{I}$ conjunctive) and impose a constraint $\phi_{\tilde{I}}$ that $\tilde{I}$ is an inductive invariant for $\mathcal{T}$.
- Fix a template for the propositional predicate map BI of type-(c, d) and impose a constraint $\phi_{BI, R^{NA}}$ that BI is an inductive backward invariant for $\mathcal{T}^{r, \tilde{I}(\ell_{out})}_{R^{NA}}$.
- For each transition τ in $\mathcal{T}$, the constraint ϕ_τ encodes non-inductiveness of BI with respect to τ in $\mathcal{T}$:

$$\mathbf{x}, \mathbf{x}' \models BI(l) \wedge \rho_\tau \wedge \neg BI(l').$$

The algorithm then solves $\Phi_2 = \phi_{\tilde{I}} \wedge \phi_{BI, R^{NA}} \wedge \bigvee_{\tau \in \mapsto} \phi_\tau$ by using an SMT-solver. If a solution is found, the algorithm uses an off-the-shelf safety prover to check if there exists a configuration in $\neg BI$ reachable in $\mathcal{T}$. Such configuration is then diverging w.r.t. $\mathcal{T}_{R^{NA}}$, so we report non-termination.

The pseudocode for our algorithm is shown in Algorithm 1. The following theorem proves soundness of our algorithm, and its proof can be found in the extended version of the paper [11].

Theorem 5.3 (Soundness)**.** *If Algorithm 1 outputs "Non-termination" for some program P, then P is non-terminating.*

Remark 2 (Algorithm termination). Our algorithm might not always terminate because either the employed SMT-solver or the safety prover might diverge. Thus, in practice one needs to impose a timeout in order to ensure algorithm termination.

5.3 Demonstration on Examples

We demonstrate our algorithm on two examples illustrating the key aspects. In the extended version [11], we present an example demonstrating an application of our method on program whose all non-terminating traces are aperiodic.

Example 5.4. Consider again our running example in Fig. 1. We demonstrate that Check 1 of our algorithm can prove that it is non-terminating. Define the resolution of non-determinism R^{NA} to assign a constant expression 9 to the

```
n := 0, b := 0, u := 0
l0 : while b == 0 and n ≤ 99 do
l1 :       u := ndet()
l2 :       if u ≤ −1 then
l3 :            b := −1
           else if u == 0 then
l4 :            b := 0
l5 :       else b := 1 fi
l6 :       n := n + 1
l7 :       if n ≥ 100 and b ≥ 1 then
l8 :            while true do
l9 :              skip
     od fi od
```

Figure 2. An example of a program without an initial diverging configuration with respect to any resolution of non-determinism that uses polynomials of degree less than 100, but for which Check 2 proves non-termination.

non-deterministic assignment, an initial configuration $\mathbf{c} = (\ell_{init}, 9, 0)$, and a propositional predicate map I as $I(\ell) = (x \geq 9)$ for $\ell \neq \ell_{out}$ and $I(\ell_{out}) = \emptyset$. Then I is an inductive invariant for $\mathcal{T}_{R^{NA}}$ with the initial configuration $\mathbf{c}$. Thus the system of polynomial constraints constructed by Check 1 is feasible, proving that this program is non-terminating.

Example 5.5. Consider the program in Fig. 2. Its initial variable valuation is given by the assertion $(n = 0 \wedge b = 0 \wedge u = 0)$, and a program execution is terminating so long as it does not assign 0 to u in the first 99 iterations of the outer loop, and then at least 1 in the 100-th iteration. Thus, if the initial configuration was diverging with respect to a resolution of non-determinism which resolves the non-deterministic assignment of u by a polynomial $p(n, b, u)$, this polynomial would need to satisfy $p(n, 0, 0) = 0$ for $n = 0, 1, \ldots, 98$ and $p(99, 0, 0) \geq 1$. Hence, the degree of p would have to be at least 100, and this program has no initial diverging configuration with respect to any resolution of non-determinism that is feasible to compute by using the Check 1 of our algorithm.

We now show that Check 2 can prove non-termination of this program using only polynomials of degree 0, i.e. constant polynomials. Define R^{NA}, Θ, BI and τ as follows:

- R^{NA} assigns constant expression 1 to the assignment of u at ℓ_1;
- $\tilde{I}(\ell) = (0 \leq n \leq 100)$ for each location ℓ;
- BI is a propositional predicate map defined via

$$BI(\ell) = \begin{cases} (0 \leq n \leq 100) & \text{if } \ell = \ell_{out} \\ (n \leq 100) & \text{if } \ell = \ell_0 \\ (n \leq 99) \vee (n = 100 \wedge b \leq 0) & \text{if } \ell = \ell_7 \\ (n \leq 98) \vee (n = 99 \wedge b \leq 0) & \text{if } \ell = \ell_6 \\ (n \leq 98) & \text{if } \ell \in \{\ell_1, \ell_5\} \\ (n \leq 99) & \text{if } \ell \in \{\ell_3, \ell_4\} \\ (n \leq 98) \vee (n = 99 \wedge u \leq 0) & \text{if } \ell = \ell_2 \\ (1 \leq 0) & \text{if } \ell \in \{l_8, l_9\}; \end{cases}$$

- τ is the transition from ℓ_0 to ℓ_1.

To show that these R^{NA}, $\tilde{I}$, BI and τ satisfy each condition in Check 2, we note that:

(1) The set of variable valuations reachable in the program upon termination is $(n, b) \in \{(n, b) \mid 1 \leq n \leq 99 \wedge b! = 0\} \cup \{(100, b) \mid b \leq 0\}$, thus $\Theta = \tilde{I}(\ell_{out})$ contains it;
(2) BI is an inductive backward invariant for $\mathcal{T}^{r,\tilde{I}(\ell_{out})}_{R^{NA}}$ (which can be checked by inspection of the reversed transition system in the extended version of the paper [11]);
(3) BI is not inductive w.r.t. τ in $\mathcal{T}$, since $(99, 0, 0) \in BI(\ell_0)$ but the variable valuation $(99, 0, 0)$ obtained by executing τ in $\mathcal{T}$ is not contained in $BI(\ell_1)$.

Thus, these R^{NA}, $\tilde{I}$, BI and τ present a solution to the system of constraints defined by Check 2. Since the configuration $(\ell_1, 99, 0, 0)$ is reachable in this program by assigning $u := 0$ in the first 99 iterations of the outer loop, but $(99, 0, 0) \notin BI(\ell_1)$, the safety prover will be able to show that a configuration in $\neg BI$ is reachable. Hence our algorithm is able to prove non-termination.

5.4 Relative Completeness

At the beginning of Section 5 we noted that constraint solving-based inductive invariant generation is relatively complete [10, 15, 29, 42, 43], in the sense that whenever there is an inductive invariant representable using the given template, the algorithm will find such an invariant. This means that our algorithm is also relatively complete in checking whether the program satisfies properties encoded as polynomial constraints in Check 1 and Check 2. Since successful Check 1 does not require a subsequent call to a safety prover, it provides to the best of our knowledge the first *relatively complete algorithm* for proving non-termination of programs with polynomial integer arithmetic and non-determinism.

Theorem 5.6 (Relative completeness)**.** *Let P be a program with polynomial integer arithmetic and $\mathcal{T}$ its transition system. Suppose that $\mathcal{T}$ admits a proper under-approximation U which restricts each non-deterministic assignment to a polynomial assignment, and a propositional predicate map C which is a closed recurrence set in U. Then for sufficiently high values of parameters c, d and D bounding the template size for invariants*

and the maximal polynomial degree, our algorithm proves non-termination of the program P.

While relative completeness guarantees in Theorem 5.6 are the first such guarantees for programs with non-determinism, they only apply to non-terminating programs that contain an initial diverging configuration w.r.t. some resolution of non-determinism. However, Example 5.5 shows that finding such a configuration might require using very high degree polynomials to resolve non-determinism, and in general such a configuration need not exist at all in non-terminating programs. In order to ensure catching non-termination bugs in such examples, an algorithm with stronger guarantees is needed. To that end, we propose a modification of our algorithm for programs in which non-determinism appears only in branching. The new algorithm provides *stronger relative completeness guarantees* that can detect non-terminating behavior in programs with no initial diverging configurations or for which Check 1 is not practical, including the program in Example 5.5 (that is, its equivalent version in which non-determinism appears only in branching as we demonstrate in Example 5.8).

To motivate this modification, let us look back at the conditions imposed on the predicate map BI by our algorithm. BI is required not to be an invariant, so that $\neg BI$ contains a reachable configuration. However, this reachability condition cannot be encoded using polynomial constraints, so instead we require that $\neg BI$ is not an inductive invariant, and then employ a safety prover which does not provide any guarantees. Our modification is based on the recent work of [1], which presents a relatively complete method for reachability analysis in polynomial programs with non-determinism appearing only in branching.

Relatively complete reachability analysis. We give a high level description of the method in [1]. Let P be a program with non-determinism appearing only in branching, $\mathcal{T}$ its transition system, and C a set of configurations defined by a propositional predicate map. The goal of the analysis is to check whether some configuration in C is reachable in $\mathcal{T}$.

The witness for the reachability of C in [1] consists of (1) an initial configuration $\mathbf{c}$, (2) a propositional predicate map $C^\diamond$ that contains $\mathbf{c}$, and (3) a polynomial ranking function f^C for $C^\diamond$ with respect to C. A *polynomial ranking function* for $C^\diamond$ with respect to C is a map f^C that to each location $\ell \in L$ assigns a polynomial expression $f^C(l)$ over program variables, such that each configuration $(l, \mathbf{x}) \in C^\diamond \backslash C$ has a successor $(l', \mathbf{x}') \in C^\diamond$ with

$$f^C(l)(\mathbf{x}) \geq f^C(l')(\mathbf{x}') + 1 \ \wedge \ f^C(l)(\mathbf{x}) \geq 0,$$

where $C^\diamond$ and C are treated as sets of configurations. Intuitively, this means that for each configuration $(l, \mathbf{x}) \in C^\diamond \backslash C$, the value of f^C at this configuration is non-negative and there is a successor of this configuration in $C^\diamond$ at which the value of f^C decreases by at least 1. If the program admits such a witness, then we may exhibit a path from $\mathbf{c}$ to a configuration in C by inductively picking either a successor in C (and thus proving reachability), or a successor in $C^\diamond \backslash C$ along which f^C decreases by 1. As the value of f^C in $\mathbf{c}$ is finite and f^C is non-negative on $C^\diamond \backslash C$, decrease can happen only only finitely many times and eventually we will have to pick a configuration in C. It is further shown in [1] that any reachable C admits a witness in the form of an initial configuration, a predicate map and a (not necessarily polynomial) ranking function.

For programs with non-determinism appearing only in branching, it is shown in [1] that all the defining properties of $\mathbf{c}$, $C^\diamond$ and f^C can be encoded using polynomial constraints. Thus [1] searches for a reachability witness by introducing template variables for $\mathbf{c}$, $C^\diamond$ and f^C, encoding the defining properties using polynomial constraints and then reducing to constraint solving. The obtained constraints are at most quadratic in the template variables, as was the case in our algorithm for proving non-termination. Moreover, their analysis is relatively complete - if a witness of reachability in the form of an initial configuration $\mathbf{c}$, a propositional predicate map $C^\diamond$ and a polynomial ranking function f^C exists, the method of [1] will find it.

Modification of our algorithm. The modified algorithm is similar to Check 2, with only difference being that we encode reachability of $\neg BI$ using polynomial constraints instead of requiring it not to be inductive in $\mathcal{T}$. The algorithm introduces a template of fresh variables determining R^{NA}, $\tilde{I}$ and BI. In addition, it introduces a template of fresh variables determining an initial configuration $\mathbf{c}$, a propositional predicate map $C^\diamond$ and a polynomial ranking function $f^{\neg BI}$. The algorithm then imposes the following polynomial constraints:

- Encode the same conditions on R^{NA}, $\tilde{I}$ and BI as in Check 2 to obtain $\Phi_{backward}$.
- Introduce fresh variables $c_1, c_2, \ldots, c_{|\mathcal{V}|}$ defining the variable valuation of $\mathbf{c}$. Then substitute these variables into the assertion Θ_{init} specifying initial configurations in $\mathcal{T}$ to obtain the constraint $\phi_{\mathbf{c}}$ for $\mathbf{c}$ being an initial configuration.
- Fix a template for the propositional predicate map $C^\diamond$ of type-(c, d) and maximal polynomial degree D. Encode that $C^\diamond$ contains $\mathbf{c}$ into the constraint $\phi_{\mathbf{c}, C^\diamond}$.
- For each location ℓ in $\mathcal{T}$, fix a template for a polynomial $f^{\neg BI}(\ell)$ over program variables of degree at most D. That is, introduce a fresh template variable for each coefficient of such a polynomial.
- Using the method of [1], for each locaiton ℓ encode the following condition

$$\forall \mathbf{x}.\mathbf{x} \models C^\diamond(\ell) \Rightarrow \mathbf{x} \in \neg BI(\ell) \vee \Big(\Big(\bigvee_{\tau=(\ell,\ell',\rho_\tau)} \mathbf{x}' \models C^\diamond(\ell') \wedge$$

$$\rho_\tau(\mathbf{x}, \mathbf{x}') \wedge f^{\neg BI}(\ell)(\mathbf{x}) \geq f^{\neg BI}(\ell')(\mathbf{x}') + 1 \Big) \ \wedge \ f^{\neg BI}(\ell)(\mathbf{x}) \geq 0 \Big),$$

as a polynomial constraint $\phi_{\ell,reach}$. Note that, since we assume that non-determinism appears only in branching and not in variable assignments, for any $\mathbf{x}$ there is at most one variable valuation $\mathbf{x}'$ such that $\rho_\tau(\mathbf{x},\mathbf{x}')$ is satisfied. Thus, the above condition indeed encodes the condition that, if $(\ell,\mathbf{x}) \notin \neg BI$, then at least one successor configuration satisfies the ranking function property. It is shown in [1] that this condition can be encoded into existentially quantified polynomial constraints over template variables, by using analogous semi-algebraic techniques that are used for inductive invariant generation in [10, 15] and which we use for invariant synthesis. We then take $\Phi_{reach} = \wedge_\ell \phi_{\ell,reach}$.

The algorithm then tries to solve $\Phi_{modified} = \Phi_{backward} \wedge \phi_{\mathbf{c}} \wedge \phi_{\mathbf{c},C^\diamond} \wedge \Phi_{reach}$.

Soundness of the modified algorithm follows the same argument as the proof of Theorem 5.3. The following theorem presents the stronger relative completeness guarantees provided by the modified algorithm.

Theorem 5.7 (Stronger relative completeness)**.** *Let P be a program with polynomial integer arithmetic, in which non-determinism appears only in branching. Let $\mathcal{T}$ be its transition system. Suppose that $\mathcal{T}$ admits*

1. *a proper under-approximation U restricting each non-deterministic assignment to a polynomial assignment,*
2. *a propositional predicate map $\tilde{I}$ which is an inductive invariant in $\mathcal{T}$,*
3. *a propositional predicate map BI which is an inductive backward invariant in $\mathcal{T}_U^{r,\tilde{I}(\ell_{out})}$, and*
4. *a witness of reachability of $\neg BI$ as in [1].*

Then for high enough values of c, d and D bounding the template size for invariants and the polynomial degree, our algorithm proves non-termination of the program P.

Remark 3. The method of [1] encodes constraints for programs in which non-determinism appears only in branching, whereas in this work we talked about constraint encoding for programs in which non-determinism appears only in assignments. This is not an issue in the modified algorithm - we can always start with a program in which non-determinism appears only in branching to encode the reachability witness constraints, and then apply the trick from Section 2 to replace each non-deterministic branching by an assignment.

Example 5.8. We show that the relative completeness guarantees of the modified algorithm apply to the program obtained from Fig. 2 by replacing the non-deterministic assignment of u and the subsequent conditional branching with the non-deterministic branching given by **if** $*$ **then**. Specifically, the new program is obtained by removing the non-deterministic assignment of u from the program, merging ℓ_1 and ℓ_2 in Fig. 5.5 into the new location $\ell_{1,2}$ and replacing the conditional by the non-deterministic branching. The reachability constraints for the modified algorithm are then encoded with respect to this new program. On the other hand, to encode the constraints as in Check 2, we consider the original program in Fig. 2.

To see that this program satisfies the conditions of Theorem 5.7, we define R^{NA}, $\tilde{I}$ and BI as in Example 5.5. Then, one witness of reachability of $\neg BI$ (where we identify $\ell_{1,2}$ with ℓ_1) is defined by $\mathbf{c} = (\ell_{init}, 0, 0, 0)$,

$$C^\diamond(\ell) = \begin{cases} (0 \le n \le 99 \wedge b = 0 \wedge u = 0) & \text{if } \ell \in \{\ell_0, \ell_{1,2}\} \\ (0 \le n \le 98 \wedge b = 0 \wedge u = 0) & \text{if } \ell \in \{\ell_4, \ell_6\} \\ (1 \le n \le 99 \wedge b = 0 \wedge u = 0) & \text{if } \ell = \ell_7 \\ (1 \le 0) & \text{otherwise;} \end{cases}$$

and

$$f^{\neg BI}(\ell, n, b, u) = \begin{cases} 5 \cdot (100 - n) + 3 & \text{if } \ell = \ell_0 \\ 5 \cdot (100 - n) + 2 & \text{if } \ell = \ell_{1,2} \\ 5 \cdot (100 - n) + 1 & \text{if } \ell = \ell_4 \\ 5 \cdot (100 - n) + 0 & \text{if } \ell = \ell_6 \\ 5 \cdot (100 - n) + 4 & \text{if } \ell = \ell_7 \\ 0 & \text{otherwise;} \end{cases}$$

To see that this is indeed the witness of reachability of $\neg BI$, observe $C^\diamond$ contains precisely the set of all configurations along the path from $\mathbf{c} = (\ell_{init}, 0, 0, 0)$ to the configuration $(\ell_1, 99, 0, 0)$ in $\neg BI$ that we described in Example 5.5 (recall, for reachability analysis we identify ℓ_1 with $\ell_{1,2}$ in the modified program in which non-determinism appears only in branching), and that $f^{\neg BI}$ is non-negative along this path and decreases by exactly 1 in each step along the path.

6 Experiments

We present a prototype implementation of our algorithm in our tool RevTerm. Our implementation is available at https://github.com/ekgma/RevTerm.git. We follow a standard approach to invariant generation [10, 15, 29] which only fixes predicate map templates at cutpoint locations. For safety prover we use CPAchecker [3] and for constraint solving we use three SMT-solvers: Barcelogic 1.2 [4], MathSAT5 [14] and Z3 [22].

Since non-determinism in all our benchmarks appears in variable assignments only, we implemented only our main algorithm and not the modified algorithm with stronger guarantees for programs with branching-only non-determinism.

Benchmarks. We evaluated our approach on benchmarks from the category *Termination of C-Integer Programs* of the Termination and Complexity Competition (TermComp'19 [26]). The benchmark suite consists of 335 programs with non-determinism: 111 non-terminating, 223 terminating, and the Collatz conjecture for which termination is unknown. We compared RevTerm against the best state-of-the-art tools that participated in this category, namely AProVE [25], Ultimate [13], VeryMax [5], and also LoAT [24].

Table 1. Experimental results with evaluation performed on the first platform. The NO/YES/MAYBE rows contain the total number of benchmarks which were proved non-terminating, terminating, or for which the tool proved neither, respectively. The next row contains the number of benchmarks proved to be non-terminating only by the respective tool. We also report the average and standard deviation (std. dev.) of runtimes. The last two rows show the runtime statistics limited to successful non-termination proofs.

	RevTerm	Ultimate	VeryMax
NO	107	97	103
YES	0	209	213
MAYBE	228	29	19
Unique NO	3	1	0
Avg. time	1.2s	5.0s	3.7s
Std. dev.	3.0s	3.7s	7.3s
Avg. time for NO	1.2s	4.4s	10.6s
Std. dev. for NO	3.0s	3.8s	9.4s

Configurations of our tool. Recall that our algorithm is parameterized by the template size for propositional predicate maps and the maximal polynomial degree. Also, it performs two checks which can be run sequentially or in parallel. Thus a *configuration* of RevTerm is defined by (a) the choice of whether we are running Check 1 or Check 2, (b) the template size (c, d) for propositional predicate maps and the maximal polynomial degree D, and (c) the choice of an SMT-solver. Our aim is to compare our algorithm to other existing approaches to non-termination proving and demonstrate generality of its relative completeness guarantees, rather than develop an optimized tool. Hence we test each configuration separately and count the total number of benchmarks that were proved to be non-terminating by at least one of the configurations. We consider configurations for both checks, each of the three SMT-solvers, and all template sizes in the set $\{(c, d, D) \mid 1 \leq c \leq 5, 1 \leq d \leq 5, 1 \leq D \leq 2\}$.

Experimental results. Our experiments were run on two platforms, and we include the results for each of them in separate tables. The first platform is Debian, 128 GB RAM, Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz, 12 Threads. The experimental results are presented in Table 1 and the timeout for each experiment was 60s.

We could not install the dependencies for AProVE on the first platform and LoAT does not support the input format of benchmarks, so we also evaluate all tools except for LoAT on StarExec [44] which is a platform on which TermComp'19 was run. We take the results of the evaluation of LoAT on StarExec from [24] which coupled it with AProVE for conversion of benchmarks to the right input format. Note however that the solver Barcelogic 1.2 is not compatible with StarExec so the number of non-terminations RevTerm proves is smaller compared to Table 1. The experimental results are presented in Table 2, and the timeout for each experiment was 60s. The timeout in both cases is on wallclock time and was chosen to match that in [24]. We note that in TermComp'19 the timeout was 300s and Ultimate proved 100 non-terminations, whereas AProVE and VeryMax proved the same number of non-terminations as in Table 2.

From Tables 1 and 2 we can see that RevTerm outperforms other tools in terms of the number of proved non-terminations. The average time for RevTerm is computed by taking the fastest successful configuration on each benchmark, so the times indicate that running multiple configurations in parallel would outperform the state-of-the-art. Since AProVE, Ultimate and VeryMax attempt to prove either termination or non-termination of programs, we include both their average times for all solved benchmarks and for non-termination proofs only.

Performance by configuration. We now discuss the performance of each configuration based on whether it runs Check 1 or Check 2 and based on which SMT-solver it uses. For the purpose of this comparison we only consider evaluation on the first platform which supports Barcelogic 1.2. Comparison of configurations in terms of the total number of solved benchmarks is presented in Table 3. We make two observations:

- Configurations using Check 1 prove 103 out of 112 non-terminations, which matches the performance of all other tools. This means that the relative completeness guarantees provided by our approach are quite general.
- Even though some SMT-solvers perform well and solve many benchmarks, none of them reaches the number 107. This means that our performance is dependent on the solver choice and designing a successful tool would possibly require multiple solvers. For example, from our results we observed that MathSAT5 performs particularly well for Check 1 with templates of small size ($c, d \in \{1, 2\}$), while Barcelogic 1.2 is best suited for templates of larger size (with $c \geq 3$) and for Check 2. While this could be seen as a limitation of our approach, it also implies that our algorithm would become even more effective with the improvement of SMT-solvers.

Finally, in the extended version [11] we present a comparison of configurations based on the template sizes for propositional predicate maps. A key observation there is that for any benchmark that RevTerm proved to be non-terminating, it was sufficient to use a template for predicate maps with $c \leq 3, d \leq 2$ and $D \leq 2$. This implies that with a smart choice of configurations, it suffices to run a relatively small number of configurations which if run in parallel would result in a tool highly competitive with the state-of-the-art.

Table 2. Experimental results with evaluation performed on StarExec [44]. The meaning of data is the same as in Table 1.

	RevTerm	LoAT	AProVE	Ultimate	VeryMax
NO	103	96	99	97	102
YES	0	0	216	209	212
MAYBE	232	239	20	29	21
Unique NO	2	1	0	0	0
Avg. time	1.8s	2.6s	4.2s	7.4s	3.8s
Std. dev.	6.6s	0.9s	4.1s	4.9s	7.4s
Avg. time NO	1.8s	2.6s	5.0s	7.0s	10.8s
Std. dev. NO	6.6s	0.9s	3.9s	7.1s	9.5s

Table 3. Comparison of configurations based on which check they run and the SMT-solver used.

	Barcelogic 1.2	MathSAT5	Z3	Total
Check 1	84	98	80	103
Check 2	69	54	63	74
Total	96	98	82	107

7 Related Work

Non-termination proving. A large number of techniques for proving non-termination consider *lasso-shaped* programs, which consist of a finite prefix (or stem) followed by a single loop without branching [30, 39]. Such techniques are suitable for being combined with termination provers [31]. Many modern termination provers repeatedly generate traces which are then used to refine the termination argument in the form of a ranking function, either by employing safety provers [19] or by checking emptiness of automata [32]. When refinement is not possible, a trace is treated like a lasso program and the prover would try to prove non-termination. However, lassos are not sufficient to detect *aperiodic* non-termination, whereas our approach handles it. Moreover, programs with nested loops typically contain infinitely many lassos which may lead to divergence, and such methods do not provide relative completeness guarantees.

TNT [30] proves non-termination by exhaustively searching for candidate lassos. For each lasso, it searches for a *recurrence set* (see Section 4) and this search is done via constraint solving. The method does not support non-determinism.

Closed recurrence sets (see Section 4) are a stronger notion than the recurrence sets, suited for proving non-termination of non-deterministic programs. The method for computing closed recurrent sets in [12] was implemented in T2 and it uses a safety prover to eliminate terminating paths iteratively until it finds a program under-approximation and a closed recurrence set in it. The method can detect aperiodic non-termination. However it is likely to diverge in the presence of many loops, as noted in [37].

The method in [37] was implemented in VeryMax [5] and it searches for witnesses to non-termination in the form of quasi-invariants, which are sets of configurations that cannot be left once they are entered. Their method searches for a quasi-invariant in each strongly-connected subgraph of the program by using Max-SMT solving. Whenever a quasi-invariant is found, safety prover is used to check its reachability. The method relies on multiple calls to a safety prover and does not provide relative completeness guarantees.

AProVE [25] proves non-termination of Java programs [9] with non-determinism. It uses constraint solving to find a recurrence set in a given loop, upon which it checks reachability of the loop. The key limitation of this approach is that for programs with nested loops for which the loop condition is not a loop invariant, it can only detect recurrence sets with a single variable valuation at the loop head.

An orthogonal approach to recurrence sets was presented in [39]. It considers lasso-shaped programs with linear arithmetic and represents infinite runs as geometric series. Their method provides relative completeness guarantees for the case of deterministic lasso-shaped programs. It also supports non-determinism, but does not provide relative completeness guarantees. The method has been implemented as a non-termination prover for lasso traces in Ultimate [13].

The method in [45] tries to prove either termination or non-termination of programs with non-determinism by making multiple calls to a safety prover. For each loop, a termination argument is incrementally refined by using a safety prover to sample a terminating trace that violates the argument. Once such terminating traces cannot be found, a safety prover is again used to check the existence of non-terminating traces in the loop.

The work of [29] considers deterministic programs with linear integer arithmetic. They present a constraint solving-based method for finding the *weakest liberal precondition (w.l.p.)* of a fixed propositional predicate map template. They then propose a method for proving non-termination which computes the w.l.p. for the postcondition "false", and then checks if it contains some initial configuration. While this approach is somewhat similar to Check 1, encoding and solving the *weakest* precondition constraints of a given template is computationally expensive and unnecessary for the purpose of proving non-termination. In Check 1, we do not impose such a strict condition. Moreover, initial diverging configurations are not sufficient to prove non-termination of non-deterministic programs. It is not immediately clear how one could use w.l.p. calculus to find a diverging configuration within a loop, like in Example 5.5.

The tool Invel [46] proves non-termination of Java programs using constraint solving and heuristics to search for recurrence sets. It only supports deterministic programs. In [38] a Hoare-style approach is developed to infer sufficient preconditions for terminating and non-terminating behavior of programs. As the paper itself mentions, the approach is not suitable for programs with non-determinism.

While all of the methods discussed above are restricted to programs with linear arithmetic, the following two methods also consider non-linear programs.

The tool Anant [18] proves non-termination of programs with non-linear arithmetic and heap-based operations. They define *live abstractions*, which over-approximate a program's transition relation while keeping it sound for proving non-termination. Their method then over-approximates non-linear assignments and heap-based commands with non-deterministic linear assignments using heuristics to obtain a live abstraction with only linear arithmetic. An approach similar to [30] but supporting non-determinism is then used, to exhaustively search for lasso traces and check if they are non-terminating. The over-approximation heuristic they present is compatible with our approach and could be used to extend our method to support operations on the heap.

LoAT [24] proves non-termination of integer programs by using loop acceleration. If a loop cannot be proved to be non-terminating, the method tries to accelerate it in order to find paths to other potentially non-terminating loops.

8 Conclusion and Future Work

We present a new approach for proving non-termination of polynomial programs with a relative completeness guarantee. For programs that do not satisfy this guarantee, our approach requires safety provers. An interesting direction of future work would be to consider approaches that can present stronger completeness guarantees. Another interesting direction would be to consider usefulness of the program reversal technique to studying other properties in programs.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. This research was partially supported by the ERC CoG 863818 (ForM-SMArt) and the Czech Science Foundation grant No. GJ19-15134Y.

References

[1] Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2020. Inductive Reachability Witnesses. *CoRR* abs/2007.14259 (2020). arXiv:2007.14259 https://arxiv.org/abs/2007.14259

[2] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. 1–3.

[3] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[4] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2008. The Barcelogic SMT Solver. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. 294–298. https://doi.org/10.1007/978-3-540-70545-1_27

[5] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2017. Proving Termination Through Conditional Termination. In *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*. 99–117. https://doi.org/10.1007/978-3-662-54577-5_6

[6] François Bourdoncle. 1993. Abstract Debugging of Higher-Order Imperative Languages. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 46–55. https://doi.org/10.1145/155090.155095

[7] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. 491–504.

[8] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better Termination Proving through Cooperation. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 413–429.

[9] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. 2011. Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*. 123–141. https://doi.org/10.1007/978-3-642-31762-0_9

[10] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 672–687. https://doi.org/10.1145/3385412.3385969

[11] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Đorđe Žikelić. 2021. Proving Non-termination by Program Reversal. arXiv:2104.01189

[12] Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. 156–171.

[13] Yu-Fang Chen, Matthias Heizmann, Ondrej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. 2018. Advanced automata-based algorithms for program termination checking. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 135–150. https://doi.org/10.1145/3192366.3192405

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of TACAS (LNCS, Vol. 7795)*, Nir Piterman and Scott Smolka (Eds.). Springer.

[15] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. 420–432.

[16] Michael Colón and Henny Sipma. 2001. Synthesis of Linear Ranking Functions. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. 67–81.

[17] Michael Colón and Henny Sipma. 2002. Practical Methods for Proving Program Termination. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. 442–454.

[18] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2014. Disproving termination with overapproximation. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. 67–74.

[19] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. 415–426.

[20] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 47–61.

[21] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*. 1–24.

[22] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[23] Nissim Francez, Orna Grumberg, Shmuel Katz, and Amir Pnueli. 1985. Proving Termination of Prolog Programs. In *Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings*. 89–105.

[24] Florian Frohn and Jürgen Giesl. 2019. Proving Non-Termination via Loop Acceleration. In *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*. 221–230.

[25] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *J. Autom. Reasoning* 58, 1 (2017), 3–31. https://doi.org/10.1007/s10817-016-9388-y

[26] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. 2019. The Termination and Complexity Competition. In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. 156–166. https://doi.org/10.1007/978-3-030-17502-3_10

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 213–223.

[28] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. 2006. SYNERGY: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*. 117–127.

[29] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 281–292.

[30] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 147–158.

[31] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*. 304–319. https://doi.org/10.1007/978-3-642-15769-1_19

[32] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 797–813. https://doi.org/10.1007/978-3-319-08867-9_53

[33] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. 58–70.

[34] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. 2018. Polynomial Invariants for Affine Programs. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. 530–539. https://doi.org/10.1145/3209108.3209142

[35] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 248–262.

[36] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear reasoning for invariant synthesis. *PACMPL* 2, POPL (2018), 54:1–54:33.

[37] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2014. Proving Non-termination Using Max-SMT. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 779–796.

[38] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and non-termination specification inference. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 489–498. https://doi.org/10.1145/2737924.2737993

[39] Jan Leike and Matthias Heizmann. 2018. Geometric Nontermination Arguments. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*. 266–283.

[40] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. 239–251.

[41] Andreas Podelski and Andrey Rybalchenko. 2004. Transition Invariants. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. 32–41.

[42] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic generation of polynomial loop. In *Symbolic and Algebraic Computation, International Symposium ISSAC 2004, Santander, Spain, July 4-7, 2004, Proceedings*. 266–273.

[43] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.* 64, 1 (2007), 54–75.

[44] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. 2014. StarExec: A Cross-Community Infrastructure for Logic Solving. In *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*. 367–373. https://doi.org/10.1007/978-3-319-08587-6_28

[45] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 54–70. https://doi.org/10.1007/978-3-662-49674-9_4

[46] Helga Velroyen and Philipp Rümmer. 2008. Non-termination Checking for Imperative Programs. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*. 154–170.