Proving Non-termination Using Max-SMT[⋆]

Daniel Larraz[1], Kaustubh Nimkar[2], Albert Oliveras[1],
Enric Rodríguez-Carbonell[1], and Albert Rubio[1]

[1] Universitat Politècnica de Catalunya, Barcelona
[2] University College London

Abstract. We show how Max-SMT-based invariant generation can be exploited for proving non-termination of programs. The construction of the proof of non-termination is guided by the generation of *quasi-invariants* – properties such that if they hold at a location during execution once, then they will continue to hold at that location from then onwards. The check that quasi-invariants can indeed be reached is then performed separately. Our technique considers strongly connected subgraphs of a program's control flow graph for analysis and thus produces more generic witnesses of non-termination than existing methods. Moreover, it can handle programs with unbounded non-determinism and is more likely to converge than previous approaches.

1 Introduction

While the problem of proving program termination has now been extensively studied [1–22], relatively less work has been done on proving non-termination of programs.

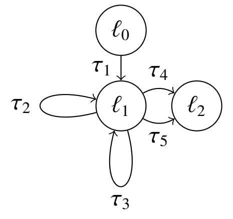
In this paper we present a new method for proving non-termination of sequential non-deterministic programs that leverages Max-SMT-based invariant generation [23, 24]. Our method analyses each *Strongly Connected SubGraph (SCSG)* of a program's control flow graph and, by means of Max-SMT solving, tries to find a formula at every node of the SCSG that satisfies certain properties. First, the formula has to be a *quasi-invariant*, i.e, it must satisfy the consecution condition of inductive invariants, but not necessarily the initiation condition. Hence, if it holds at the node during execution once, then it continues to hold from then onwards. Second, the formula has to be *edge-closing*, meaning that it forbids taking any of the outgoing edges from that node that exit the SCSG. Now, once we have computed an edge-closing quasi-invariant for every node of the SCSG, if a state is reached that satisfies one of them, then program execution will remain within the SCSG from then onwards. The existence of such a state is tested with an off-the-shelf reachability checker. If it succeeds, we have proved non-termination of the original program, and the edge-closing quasi-invariants of the SCSG and the trace given by the reachability checker form the witness of non-termination.

Our approach differs from previous methods in two major ways. First, edge-closing quasi-invariants are more generic properties than non-termination witnesses produced by other provers, and thus are likely to carry more information and be more useful in bug finding. Second, our non-termination witnesses include SCSGs, which are larger structures than, e.g., *lassos*. Note that the number of SCSGs present in any CFG is finite, while the number of lassos is infinite. Because of these differences, our method is more likely to converge. Moreover, lasso-based methods can only handle periodic non-termination, while our approach can deal with aperiodic non-termination too.

⋆ Partially supported by Spanish MEC/MICINN under grant TIN 2010-21062-C02-01.

```
ℓ0: int i, j;
    j := -1;
ℓ1: while (i > 0 && j != 0)
        i := i + j;
        j := j + 2;
ℓ2:
```

(a)



$\mathcal{R}_{\tau_1}$: $j' = -1$

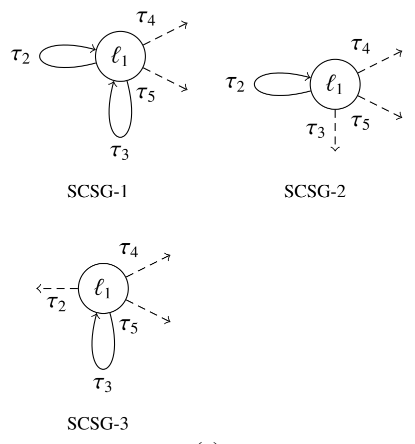
$\mathcal{R}_{\tau_2}$: $i \geq 1 \ \wedge \ j \leq -1 \ \wedge$
$i' = i + j \ \wedge \ j' = j + 2$

$\mathcal{R}_{\tau_3}$: $i \geq 1 \ \wedge \ j \geq 1 \ \wedge$
$i' = i + j \ \wedge \ j' = j + 2$

$\mathcal{R}_{\tau_4}$: $i \leq 0 \ \wedge \ i' = i \ \wedge \ j' = j$

$\mathcal{R}_{\tau_5}$: $i \geq 1 \ \wedge \ j = 0 \ \wedge \ i' = i \ \wedge \ j' = j$

(b)



(c)

For SCSG-3 :

Iteration 1 :

Solution for M_{ℓ_1}	:	$j \geq 1$
Disabled transitions	:	τ_2, τ_5
Quasi-invariant Q_{ℓ_1}	:	$j \geq 1$

Iteration 2 :

Solution for M_{ℓ_1}	:	$i \geq 1$
Disabled transitions	:	τ_4
Quasi-invariant Q_{ℓ_1}	:	$j \geq 1 \ \wedge \ i \geq 1$

Reachable path **:** $\ell_0 \to \ell_1 \to \ell_1$

(d)

Fig. 1. Example program (a) together with its corresponding CFG (b), non-trivial SCSGs (c) and non-termination analysis (d)

Our technique is based on constraint solving for invariant generation [25] and is goal-directed. Before discussing it formally, we describe it with a simple example. Consider the program in Fig. 1(a). The CFG for this program is shown in Fig. 1(b). The edges of the CFG represent the transitions between the locations. For every transition τ, we denote the formula of its transition relation by $\mathcal{R}_\tau(i, j, i', j')$. The unprimed variables represent the values of the variables before the transition, and the primed ones represent the values after the transition. By $\mathcal{R}_\tau(i, j)$ we denote the *conditional part of* τ, which only involves the pre-variables. Fig. 1(c) shows all non-trivial (i.e. with at least one edge) SCSGs present in the CFG. For every SCSG, the dashed edges are those that exit the SCSG and hence are not part of it. Note that SCSG-1 is a maximal strongly connected subgraph, and thus is a strongly connected component of the CFG. Notice also that τ_3 is an additional exit edge for SCSG-2, and similarly τ_2 is an exit edge for SCSG-3. The non-termination of this example comes from SCSG-3.

Our approach considers every SCSG of the graph one by one. In every iteration of our method, we try to find a formula at every node of the SCSG under consideration. This formula is originally represented as a template with unknown coefficients. We then form a system of constraints involving the template coefficients in the Max-SMT framework. In a Max-SMT problem, some of the constraints are *hard*, meaning that any solution to the system of constraints must satisfy them, and others are *soft*, which may or may not be satisfied. Soft constraints carry a weight, and the goal of the Max-SMT solver is to find a solution for the hard constraints such that the sum of the weights for the soft constraints violated by the solution is minimized. In our method, essentially the hard constraints encode that the formula should obey the consecution condition, and every soft constraint encodes that the formula will disable an exit edge. A solution to this system of constraints assigns values to template coefficients, thus giving us the required formula at every node.

Consider the analysis of SCSG-3 (refer to Fig. 1(d)). Note that there is a single node ℓ_1 and a single transition τ_3 in SCSG-3. We denote by $E = \{\tau_2, \tau_4, \tau_5\}$ the set of exit edges for SCSG-3. By $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ we denote the quasi-invariant at node ℓ_1. Initially $Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \triangleq \texttt{true}$. In the first iteration, for node ℓ_1 we assign a template $M_{\ell_1}(\mathsf{i}, \mathsf{j}) : a.\mathsf{i} + b.\mathsf{j} \leq c$.

We then form the Max-SMT problem consisting of the following system of hard and soft constraints:

(**Consecution**) $\forall\, \mathsf{i}, \mathsf{j}, \mathsf{i}', \mathsf{j}'.\ M_{\ell_1}(\mathsf{i}, \mathsf{j}) \ \wedge\ Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \ \wedge\ \mathcal{R}_{\tau_3}(\mathsf{i}, \mathsf{j}, \mathsf{i}', \mathsf{j}') \rightarrow M_{\ell_1}(\mathsf{i}', \mathsf{j}')$

(**Edge-Closing**) For all $\tau \in E$: $\forall\, \mathsf{i}, \mathsf{j}.\ M_{\ell_1}(\mathsf{i}, \mathsf{j}) \ \wedge\ Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \rightarrow \neg\mathcal{R}_{\tau}(\mathsf{i}, \mathsf{j})$

The consecution constraint is hard, while the edge-closing constraints are soft (with weight, say, 1). The edge-closing constraint for $\tau \in E$ encodes that, from any state satisfying $M_{\ell_1}(\mathsf{i}, \mathsf{j}) \wedge Q_{\ell_1}(\mathsf{i}, \mathsf{j})$, the transition τ is disabled and cannot be executed.

In the first iteration, a solution for M_{ℓ_1} gives us the formula $\mathsf{j} \geq 1$. This formula satisfies the edge-closing constraints for τ_2 and τ_5. We conjoin this formula to Q_{ℓ_1}, updating it to $Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \triangleq \mathsf{j} \geq 1$. We also update $E = \{\tau_4\}$ by removing τ_2 and τ_5, as these edges are now disabled.

In the second iteration, we again consider the same template $M_{\ell_1}(\mathsf{i}, \mathsf{j})$ and try to solve the Max-SMT problem above with updated $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ and E. This time we get a solution that gives us the formula $\mathsf{i} \geq 1$, which satisfies the edge-closing constraint for τ_4. We again update $Q_{\ell_1}(\mathsf{i}, \mathsf{j}) \triangleq \mathsf{j} \geq 1 \ \wedge\ \mathsf{i} \geq 1$ by conjoining the new formula. We update $E = \emptyset$ by removing the disabled edge τ_4. Now all exit edges have been disabled, and thus the quasi-invariant $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ is edge-closing.

In the final step of our method, we use a reachability checker to determine if any state satisfying $Q_{\ell_1}(\mathsf{i}, \mathsf{j})$ at location ℓ_1 is reachable. This test succeeds, and a path $\ell_0 \rightarrow \ell_1 \rightarrow \ell_1$ is obtained. Notice that the path goes through the loop once before we reach the required state. At this point, we have proved non-termination of the original program.

2 Preliminaries

2.1 SMT and Max-SMT

Let $\mathcal{P}$ be a finite set of *propositional variables*. If $p \in \mathcal{P}$, then p and $\neg p$ are *literals*. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* is a disjunction of literals. A *propositional formula* is a conjunction of clauses. The problem of *propositional satisfiability* (abbreviated as SAT) consists in, given a formula, determining whether or not it is *satisfiable*, i.e., if it has a *model*: an assignment of Boolean values to variables that satisfies the formula.

An extension of SAT is the *satisfiability modulo theories (SMT)* problem [26]: to decide the satisfiability of a given quantifier-free first-order formula with respect to a background theory. In this setting, a model (which we may also refer to as a *solution*) is an assignment of values from the theory to variables that satisfies the formula. Here we will consider the theories of *linear real/integer arithmetic (LRA/LIA)*, where literals are linear inequalities over real and integer variables respectively, and the more general theories of *non-linear real/integer arithmetic (NRA/NIA)*, where literals are polynomial inequalities over real and integer variables, respectively.

Another generalization of SAT is the *Max-SAT* problem [26]: it consists in, given a *weighted* formula where each clause has a weight (a positive number or infinity), finding the assignment such that the cost, i.e., the sum of the weights of the falsified clauses, is minimized. Clauses with infinite weight are called *hard*, while the rest are called *soft*. Equivalently, the problem can be seen as finding the model of the hard clauses such that the sum of the weights of the falsified soft clauses is minimized.

Finally, the problem of *Max-SMT*[27] merges Max-SAT and SMT, and is defined from SMT analogously to how Max-SAT is derived from SAT. Namely, the *Max-SMT* problem consists in, given a weighted formula, to find an assignment that minimizes the sum of the weights of the falsified clauses in the background theory.

2.2 Transition Systems

Our technique is applicable to sequential non-deterministic programs with integer variables and commands whose transition relations can be expressed in linear (integer) arithmetic. By $\overline{\mathsf{v}}$ we represent the tuple of program variables. For the sake of presentation, we assume that the non-determinism of programs can come only from non-deterministic assignments of the form i := `nondet`(), where $\mathsf{i} \in \overline{\mathsf{v}}$ is a program variable. Note that, however, this assumption still allows one to encode other kinds of non-determinism. For instance, any non-deterministic branching of the form **if**$(*)\{\}$ **else**$\{\}$ can be cast into this framework by introducing a new program variable $\mathsf{k} \in \overline{\mathsf{v}}$ and rewriting into the form k := `nondet`(); **if**$(\mathsf{k} \geq 0)\{\}$ **else**$\{\}$.

We model programs with *transition systems*. A transition system $\mathcal{S} = (\overline{\mathsf{v}}, \overline{\mathsf{u}}, \mathcal{L}, \Theta, \mathcal{T})$ consists of a tuple of *program variables* $\overline{\mathsf{v}}$, a tuple of *non-deterministic variables* $\overline{\mathsf{u}}$, a set of *locations* $\mathcal{L}$, a map Θ from locations to formulas characterizing the initial values of the variables, and a set of *transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is a triple $(\ell, \ell', \mathcal{R})$, where $\ell, \ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and $\mathcal{R}$ is the *transition relation*: a formula over the non-deterministic variables $\overline{\mathsf{u}}$, the program variables $\overline{\mathsf{v}}$ and their primed versions $\bar{\mathsf{v}}'$, which represent the values of the variables after the transition. The transition relation of a non-deterministic assignment of the form $\mathsf{i} := \mathtt{nondet}()$, where $\mathsf{i} \in \bar{\mathsf{v}}$, is represented by the formula $\mathsf{i}' = \mathsf{u}_1$, where $\mathsf{u}_1 \in \bar{\mathsf{u}}$ is a fresh non-deterministic variable. Note that u_1 is not a program variable, i.e., $\mathsf{u}_1 \notin \bar{\mathsf{v}}$, and is added only to model the non-deterministic assignment. Thus, without loss of generality on the kind of non-deterministic programs we can model, we will assume that every non-deterministic variable appears in at most one transition relation. A transition that includes a non-deterministic variable in its transition relation is called *non-deterministic* (abbreviated as `nondet`).

In what follows we will assume that transition relations are described as conjunctions of linear inequalities over program variables and non-deterministic variables. Given a transition relation $\mathcal{R} = \mathcal{R}(\bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}')$, we will use $\mathcal{R}(\bar{\mathsf{v}})$ to denote the *conditional part of* $\mathcal{R}$, i.e., the conjunction of linear inequalities in $\mathcal{R}$ containing only variables in $\bar{\mathsf{v}}$. For a transition system modeling real programs, the following conditions are true:

$$\text{For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T} : \forall \bar{\mathsf{v}}, \bar{\mathsf{u}}\, \exists \bar{\mathsf{v}}'.\ \mathcal{R}(\bar{\mathsf{v}}) \to \mathcal{R}(\bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}'). \tag{1}$$

$$\text{For } \ell \in \mathcal{L} : \bigvee_{(\ell,\ell',\mathcal{R})} \mathcal{R}(\bar{\mathsf{v}}) \triangleq \mathtt{true}. \tag{2}$$

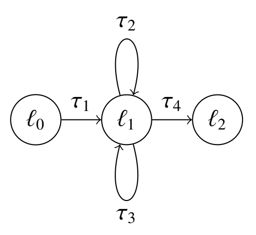
$$\text{For } \tau_1 = (\ell, \ell_1, \mathcal{R}_1), \tau_2 = (\ell, \ell_2, \mathcal{R}_2) \in \mathcal{T}, \tau_1 \neq \tau_2 : \mathcal{R}_1(\bar{\mathsf{v}}) \wedge \mathcal{R}_2(\bar{\mathsf{v}}) \triangleq \mathtt{false}. \tag{3}$$

Condition (1) guarantees that next values for the program variables always exist if the conditional part of the transition holds. Condition (2) expresses that, for any location, at least one of the outgoing transitions from that location can always be executed. Finally, condition (3) says that any two different transitions from the same location are mutually exclusive, i.e., conditional branching is always deterministic.

Example 1. Let us consider the program shown in Figure 2. Note how the two non-deterministic assignments have been replaced in the CFG by assignments to fresh non-deterministic variables u_1 and u_2. Condition (2) is trivially satisfied for ℓ_0 and ℓ_2, since the conditional part of their outgoing transition relations is empty. Regarding ℓ_1, clearly the formula $\mathsf{x} \geq \mathsf{y} \ \vee\ \mathsf{x} < \mathsf{y}$ is a tautology. Condition (3) is also easy to check: the conditional parts of $\mathcal{R}_{\tau_2}, \mathcal{R}_{\tau_3}$ and $\mathcal{R}_{\tau_4}$ are pairwise unsatisfiable. Finally, condition (1) trivially holds since the primed parts of the transition relations consist of equalities whose left-hand side is always a different variable. $\square$

A *state* is an assignment of a value to each of the variables in $\bar{\mathsf{v}}$ and $\bar{\mathsf{u}}$. A *configuration* is a pair (ℓ, σ) consisting of a location $\ell \in \mathcal{L}$ and a state σ. For any pair of configurations (ℓ, σ) and (ℓ', σ'), if there is a transition $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}$ such that $(\sigma, \sigma') \models \mathcal{R}$, we write $(\ell, \sigma) \xrightarrow{\tau} (\ell', \sigma')$. A *computation* is a sequence of configurations (ℓ_0, σ_0), (ℓ_1, σ_1), ... such that $\sigma_0 \models \Theta(\ell_0)$, and for each pair of consecutive configurations there exists $\tau_i \in \mathcal{T}$ satisfying $(\ell_i, \sigma_i) \xrightarrow{\tau_i} (\ell_{i+1}, \sigma_{i+1})$. A configuration (ℓ, σ) is *reachable* if there exists a computation ending at (ℓ, σ). A transition system is *terminating* if all its computations are finite, and *non-terminating* otherwise. The goal of this paper is, given a transition system, to prove that it is non-terminating.

```
ℓ0: int x, y;
ℓ1: while (x ≥ y)
      if (x ≥ 0)
         x := nondet();
         y := y + 1;
      else
         y := nondet();
ℓ2:
```



$\mathcal{R}_{\tau_1}$: $x' = x \wedge\ y' = y$

$\mathcal{R}_{\tau_2}$: $x \geq y\ \wedge\ x \geq 0 \wedge$ $x' = u_1\ \wedge\ y' = y + 1$

$\mathcal{R}_{\tau_3}$: $x \geq y\ \wedge\ x < 0 \wedge$ $x' = x\ \wedge\ y' = u_2$

$\mathcal{R}_{\tau_4}$: $x < y\ \wedge\ x' = x \wedge$ $y' = y$

Fig. 2. Program involving non-deterministic assignments, together with its CFG

3 Quasi-invariants and Non-termination

Here we will introduce the core concept of this work, that of a *quasi-invariant*: a property such that, if it is satisfied at a location during execution once, then it continues to hold at that location from then onwards. The importance of this notion resides in the fact that it is a key ingredient in our witnesses of non-termination: if each location of an SCSG can be mapped to a quasi-invariant that is *edge-closing*, i.e., that forbids executing any of the outgoing transitions that leave the SCSG, and the SCSG can be reached at a configuration satisfying the corresponding quasi-invariant, then the program is non-terminating (if `nondet` transitions are present, additional properties are required, as will be seen below). A constructive proof of this claim is given at the end of this section.

First of all, let us define basic notation. For a strongly connected subgraph (SCSG) C of a program's CFG, we denote by $\mathcal{L}^C$ the set of locations of C, and by $\mathcal{T}^C$ the set of edges of C. We define $\mathcal{E}^C \overset{def}{=} \{\tau = (\ell, \ell', \mathcal{R}) \mid \ell \in \mathcal{L}^C, \tau \notin \mathcal{T}^C\}$ to be the set of exit edges of C.

Consider a map Q that assigns a formula $Q_\ell(\overline{v})$ to each of the locations $\ell \in \mathcal{L}^C$. Consider also a map $\mathcal{U}$ that assigns a formula $U_\tau(\overline{v}, \overline{u})$ to each transition $\tau \in \mathcal{T}^C$, which represents the *restriction* that the non-deterministic variables must obey.[3] The map Q is a *quasi-invariant map* on C with restriction $\mathcal{U}$ if:

(Consecution)

$$\text{For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C : \forall \overline{v}, \overline{u}, \overline{v}'.\ Q_\ell(\overline{v}) \wedge \mathcal{R}(\overline{v}, \overline{u}, \overline{v}') \wedge U_\tau(\overline{v}, \overline{u}) \rightarrow Q_{\ell'}(\overline{v}') \quad (4)$$

Condition (4) says that, whenever a state at $\ell \in \mathcal{L}^C$ satisfying Q_ℓ is reached and a transition from ℓ to ℓ' can be executed, then the resulting state satisfies $Q_{\ell'}$. This condition corresponds to the consecution condition for inductive invariants. Since inductive invariants are additionally required to satisfy initiation conditions [25], we refer to properties satisfying condition (4) as quasi-invariants, hence the name for Q.

[3] For the sake of presentation, we assume that U_τ is defined for all transitions, whether they are deterministic or not. In the former case, by convention U_τ is `true`.

Example 2. In order to explain the roles of $\mathcal{Q}$ and $\mathcal{U}$, consider the program in Figure 2. It is easy to see that if $\mathsf{x} \geq \mathsf{y}$ were a quasi-invariant at ℓ_1, the program would be non-terminating (provided ℓ_1 is reachable with a state such that $\mathsf{x} \geq \mathsf{y}$). However, due to the non-deterministic assignments, the property is not a quasi-invariant. On the other hand, if we add the restrictions $U_{\tau_2} := \mathsf{u}_1 \geq \mathsf{x} + 1$ and $U_{\tau_3} := \mathsf{u}_2 \leq \mathsf{y}$, which constrain the non-deterministic choices in the assignments, the quasi-invariant holds and non-termination is proved. □

Additionally, our method also needs that $\mathcal{Q}$ and $\mathcal{U}$ are *reachable* and *unblocking*:

$$\textbf{(Reachability)}\ \exists\, \ell \in \mathcal{L}^C.\ \exists\, \sigma\ \textit{s.t.}\ (\ell, \sigma) \text{ is reachable and } \sigma \models Q_\ell(\overline{\mathsf{v}}) \tag{5}$$

$$\textbf{(Unblocking)} \text{ For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C : \forall \overline{\mathsf{v}} \exists \overline{\mathsf{u}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \tag{6}$$

Condition (5) says that there exists a computation reaching a configuration (ℓ, σ) such that σ satisfies the quasi-invariant at location ℓ.

As for condition (6), consider a state σ at some $\ell \in \mathcal{L}^C$ satisfying $Q_\ell(\overline{\mathsf{v}})$. This condition says that, for any transition $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C$ from ℓ, if σ satisfies the conditional part $\mathcal{R}(\overline{\mathsf{v}})$, then we can always make a choice for the non-deterministic assignment that obeys the restriction $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$.

The last property we require from quasi-invariants is that they are edge-closing. Formally, the quasi-invariant map $\mathcal{Q}$ on C is *edge-closing* if it satisfies all of the following constraints:

$$\textbf{(Edge-Closing)} \text{ For } \tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^C : \forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \rightarrow \neg \mathcal{R}(\overline{\mathsf{v}}) \tag{7}$$

Condition (7) says that, from any state at $\ell \in \mathcal{L}^C$ that satisfies $Q_\ell(\overline{\mathsf{v}})$, all the exit transitions are disabled and cannot be executed.

The following is the main result of this section:

Theorem 1. *$\mathcal{Q}$, $\mathcal{U}$ that satisfy* (4), (5), (6) *and* (7) *for a certain SCSG C of a CFG P imply non-termination of P.*

In order to prove Theorem 1, we need the following lemma:

Lemma 1. *Let us assume that $\mathcal{Q}$, $\mathcal{U}$ satisfy* (4), (6) *and* (7) *for a certain SCSG C. Let (ℓ, σ) be a configuration such that $\ell \in \mathcal{L}^C$ and $\sigma \models Q_\ell(\overline{\mathsf{v}})$. Then there exists a configuration (ℓ', σ') such that $\ell' \in \mathcal{L}^C$, $\sigma' \models Q_{\ell'}(\overline{\mathsf{v}})$ and $(\ell, \sigma) \xrightarrow{\tau} (\ell', \sigma')$ for a certain $\tau \in \mathcal{T}^C$.*

Proof. By condition (2) (which is implicitly assumed to hold), there is a transition τ of the form $(\ell, \ell', \mathcal{R})$ for a certain $\ell' \in \mathcal{L}$ such that $\sigma \models \mathcal{R}(\overline{\mathsf{v}})$. Now, by virtue of condition (7), since $\sigma \models Q_\ell(\overline{\mathsf{v}})$ we have that $\tau \in \mathcal{T}^C$. Thus, $\ell' \in \mathcal{L}^C$. Moreover, thanks to condition (6) and $\sigma \models Q_\ell(\overline{\mathsf{v}})$ and $\sigma \models \mathcal{R}(\overline{\mathsf{v}})$, we deduce that there exist values v for the non-deterministic variables $\overline{\mathsf{u}}$ such that $(\sigma, v) \models U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$. Further, by condition (1) (which is again implicitly assumed), we have that there exists a state σ' such that $(\sigma, v, \sigma') \models \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}')$. All in all, by condition (4) and the fact that $\sigma \models Q_\ell(\overline{\mathsf{v}})$ and $(\sigma, v, \sigma') \models \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}')$ and $(\sigma, v) \models U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$, we get that $\sigma' \models Q_{\ell'}(\overline{\mathsf{v}}')$, or equivalently by renaming variables, $\sigma' \models Q_{\ell'}(\overline{\mathsf{v}})$. So (ℓ', σ') satisfies the required properties. □

```
PROVE-NT (SCSG $C$, CFG $P$)
  For $\ell \in \mathcal{L}^C$, set $Q_\ell(\bar{\mathsf{v}}) \leftarrow \mathtt{true}$
  For $\tau \in \mathcal{T}^C$, set $U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \leftarrow \mathtt{true}$
  $E^C \leftarrow \mathcal{E}^C$
  while $E^C \neq \emptyset$ do
    At $\ell \in \mathcal{L}^C$, assign a template $M_\ell(\bar{\mathsf{v}})$
    At $\tau \in \mathcal{T}^C$, assign a template $N_\ell(\bar{\mathsf{v}}, \bar{\mathsf{u}})$
    Solve Max-SMT problem with
        hard constraints (8), (9), (10) and soft constraints (11)
    if no model for hard clauses is found then return Unknown, $\perp$ fi
    For $\ell \in \mathcal{L}^C$, let $\widehat{M_\ell}(\bar{\mathsf{v}})$ = Solution for $M_\ell(\bar{\mathsf{v}})$
    For $\tau \in \mathcal{T}^C$, let $\widehat{N_\tau}(\bar{\mathsf{v}}, \bar{\mathsf{u}})$ = Solution for $N_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}})$
    For $\ell \in \mathcal{L}^C$, set $Q_\ell(\bar{\mathsf{v}}) \leftarrow Q_\ell(\bar{\mathsf{v}}) \wedge \widehat{M_\ell}(\bar{\mathsf{v}})$
    For $\tau \in \mathcal{T}^C$ set $U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \leftarrow U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \wedge \widehat{N_\tau}(\bar{\mathsf{v}}, \bar{\mathsf{u}})$
    Remove from $E^C$ disabled edges
  done
  for all $\ell \in \mathcal{L}^C$ do
    if Reachable $(\ell, \sigma)$ in $P$ s.t. $\sigma \models Q_\ell(\bar{\mathsf{v}})$ then
      let $\pi$ = reachable path to $(\ell, \sigma)$
      return Non-Terminating, $(\mathcal{Q}, \mathcal{U}, \pi)$
    fi
  done
  return Unknown, $\perp$
```

Fig. 3. Procedure PROVE-NT for proving non-termination of a program P by analyzing SCSG C

Now we are ready to prove Theorem 1:

Proof (of Theorem 1). We will construct an infinite computation, which will serve as a witness of non-termination. Thanks to condition (5), we know that there exist a location $\ell \in \mathcal{L}^C$ and a state σ such that (ℓ, σ) is reachable and $\sigma \models Q_\ell(\bar{\mathsf{v}})$. As (ℓ, σ) is reachable, there is a computation π whose last configuration is (ℓ, σ). Now, since $\mathcal{Q}$, $\mathcal{U}$ satisfy (4), (6) and (7) for C, and $\ell \in \mathcal{L}^C$ and $\sigma \models Q_\ell(\bar{\mathsf{v}})$, we can apply Lemma 1 to inductively extend π to an infinite computation of P. □

4 Computing Proofs of Non-termination

In this section we explain how proofs of non-termination are effectively computed. As outlined in Section 1, first of all we exhaustively enumerate the SCSGs of the CFG. For each SCSG C, our non-termination proving procedure PROVE-NT, which will be described below, is called. By means of Max-SMT solving, this procedure iteratively computes an unblocking quasi-invariant map $\mathcal{Q}$ and a restriction map $\mathcal{U}$ for C. If the construction is successful and eventually edge-closedness can be achieved, and moreover the quasi-invariants of C can be reached, then the synthesized $\mathcal{Q}$, $\mathcal{U}$ satisfy the properties of Theorem 1, and therefore the program is guaranteed not to terminate.

In a nutshell, the enumeration of SCSGs considers a strongly connected component (SCC) of the CFG at a time, and then generates all the SCSGs included in that SCC. More precisely, first of all the SCCs are considered according to a topological ordering in the CFG. Then, once an SCC $\mathcal{S}$ is fixed, the SCSGs included in $\mathcal{S}$ are heuristically enumerated starting from $\mathcal{S}$ itself (since taking a strictly smaller subgraph would imply discarding some transitions a priori arbitrarily), then simple cycles in $\mathcal{S}$ (as they are easier to deal with), and then the rest of SCSGs included in $\mathcal{S}$.

Then, once the SCSG C is fixed, our non-termination proving procedure Prove-NT (Fig. 3) is called. The procedure takes as input an SCSG C of the program's CFG, and the CFG itself. For every location $\ell \in \mathcal{L}^C$, we initially assign a quasi-invariant $Q_\ell(\overline{\mathsf{v}}) \triangleq \mathtt{true}$. Similarly, for every transition $\tau \in \mathcal{T}^C$, we initially assign a restriction $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \triangleq \mathtt{true}$. The set E^C keeps track of the exit edges of C that have not been discarded yet, and hence at the beginning we have $E^C = \mathcal{E}^C$. Then we iterate in a loop in order to strengthen the quasi-invariants and restrictions till $E^C = \emptyset$, that is, all the exit edges of C are disabled.

In every iteration we assign a template $M_\ell(\overline{\mathsf{v}}) \equiv m_{\ell,0} + \sum_{v\in\overline{\mathsf{v}}} m_{\ell,v} \cdot v \leq 0$ to each $\ell \in \mathcal{L}^C$. We also assign a template $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \equiv n_{\tau,0} + \sum_{v\in\overline{\mathsf{v}}} n_{\tau,v} \cdot v + \sum_{\mathsf{u}\in\overline{\mathsf{u}}} n_{\tau,\mathsf{u}} \cdot \mathsf{u} \leq 0$ to each $\tau \in \mathcal{T}^C$.[4] Then we form the Max-SMT problem with the following constraints:[5]

- For $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C$:
  $$\forall \overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}'.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}, \overline{\mathsf{u}}, \overline{\mathsf{v}}') \wedge U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \to M_{\ell'}(\overline{\mathsf{v}}') \tag{8}$$
- For $\ell \in \mathcal{L}^C$: $$\exists \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \bigvee_{\tau=(\ell,\ell',\mathcal{R})\in\mathcal{T}^C} \mathcal{R}(\overline{\mathsf{v}}) \tag{9}$$
- For $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C$:
  $$\forall \overline{\mathsf{v}} \exists \overline{\mathsf{u}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \to U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \tag{10}$$
- For $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^C$: $$\forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \to \neg\mathcal{R}(\overline{\mathsf{v}}) \tag{11}$$

The constraints (8), (9) and (10) are hard, while the constraints (11) are soft.

The Max-SMT solver finds a solution $\widehat{M_\ell}(\overline{\mathsf{v}})$ for every $M_\ell(\overline{\mathsf{v}})$ for $\ell \in \mathcal{L}^C$ and a solution $\widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ for every $N_\ell(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ for $\tau \in \mathcal{T}^C$. The solution satisfies the hard constraints and as many soft constraints as possible. In other words, it is the best solution for hard constraints that disables the maximum number of transitions. We then update $Q_\ell(\overline{\mathsf{v}})$ for every $\ell \in \mathcal{L}^C$ by strengthening it with $\widehat{M_\ell}(\overline{\mathsf{v}})$, and update $U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ for every $\tau \in \mathcal{T}^C$ by strengthening it with $\widehat{N_\tau}(\overline{\mathsf{v}}, \overline{\mathsf{u}})$. We then remove all the disabled transitions from E^C and continue the iterations of the loop with updated Q, $\mathcal{U}$ and E^C. Note that, even if none of the exit edges is disabled in an iteration (i.e. no soft constraint is met), the quasi-invariants found in that iteration may be helpful for disabling exit edges later.

When all exit transitions are disabled, we exit the loop with the unblocking edge-closing quasi-invariant map Q and the restriction map $\mathcal{U}$.

Finally, we check whether there exists a reachable configuration (ℓ, σ) such that $\ell \in \mathcal{L}^C$ and $\sigma \models Q_\ell(\overline{\mathsf{v}})$ with an off-the-shelf reachability checker. If this test succeeds,

[4] Actually templates $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ are only introduced for `nondet` transitions. To simplify the presentation, we assume that for other transitions, $N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$ is `true`.

[5] For clarity, leftmost existential quantifiers over the unknowns of the templates are implicit.

we report non-termination along with $Q, \mathcal{U}$ and the path π reaching (ℓ, σ) as a witness of non-termination.

The next theorem formally states that PROVE-NT proves non-termination:

Theorem 2. *If procedure* PROVE-NT *terminates on input SCSG C and CFG P with* Non-Terminating, *$(Q, \mathcal{U}, \pi)$, then program P is non-terminating, and $(Q, \mathcal{U}, \pi)$ allow building an infinite computation of P.*

Proof. Let us prove that, if PROVE-NT terminates with Non-Terminating, then the conditions of Theorem 1, i.e., conditions (4), (5), (6) and (7) are met.

First of all, let us prove by induction on the number of iterations of the **while** loop that conditions (4) and (6) are satisfied, and also that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^C - E^C$,

$$\forall \bar{\mathsf{v}}.\ Q_\ell(\bar{\mathsf{v}}) \rightarrow \neg\mathcal{R}(\bar{\mathsf{v}}).$$

Before the loop is executed, for all locations $\ell \in \mathcal{L}^C$ we have that $Q_\ell(\bar{\mathsf{v}}) \triangleq \mathtt{true}$ and for all $\tau \in \mathcal{T}^C$ we have that $U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \triangleq \mathtt{true}$. Conditions (4) and (6) are trivially met. The other remaining condition holds since initially $E^C = \mathcal{E}^C$.

Now let us see that each iteration of the loop preserves the three conditions. Regarding (4), by induction hypothesis we have that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C$,

$$\forall \bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}'.\ Q_\ell(\bar{\mathsf{v}}) \wedge \mathcal{R}(\bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}') \wedge U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \rightarrow Q_{\ell'}(\bar{\mathsf{v}}').$$

Moreover, the solution computed by the Max-SMT solver satisfies constraint (8), i.e., has the property that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C$,

$$\forall \bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}'.\ Q_\ell(\bar{\mathsf{v}}) \wedge \widehat{M_\ell}(\bar{\mathsf{v}}) \wedge \mathcal{R}(\bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}') \wedge U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \wedge \widehat{N_\tau}(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \rightarrow \widehat{M_{\ell'}}(\bar{\mathsf{v}}').$$

Altogether, we have that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C$,

$$\forall \bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}'.(Q_\ell(\bar{\mathsf{v}}) \wedge \widehat{M_\ell}(\bar{\mathsf{v}})) \wedge \mathcal{R}(\bar{\mathsf{v}}, \bar{\mathsf{u}}, \bar{\mathsf{v}}') \wedge (U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \wedge \widehat{N_\tau}(\bar{\mathsf{v}}, \bar{\mathsf{u}})) \rightarrow (Q_{\ell'}(\bar{\mathsf{v}}') \wedge \widehat{M_{\ell'}}(\bar{\mathsf{v}}')).$$

Hence condition (4) is preserved.

As for condition (6), the solution computed by the Max-SMT solver satisfies constraint (10), i.e., has the property that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{T}^C$,

$$\forall \bar{\mathsf{v}} \exists \bar{\mathsf{u}}.\ (Q_\ell(\bar{\mathsf{v}}) \wedge \widehat{M_\ell}(\bar{\mathsf{v}})) \wedge \mathcal{R}(\bar{\mathsf{v}}) \rightarrow (U_\tau(\bar{\mathsf{v}}, \bar{\mathsf{u}}) \wedge \widehat{N_\tau}(\bar{\mathsf{v}}, \bar{\mathsf{u}})).$$

Thus, condition (6) is preserved.

Regarding the last property, note that the transitions $\tau = (\ell, \ell', \mathcal{R}) \in E^C$ that satisfy the soft constraints (11), i.e., such that

$$\forall \bar{\mathsf{v}}.\ (Q_\ell(\bar{\mathsf{v}}) \wedge \widehat{M_\ell}(\bar{\mathsf{v}})) \rightarrow \neg\mathcal{R}(\bar{\mathsf{v}})$$

are those removed from E^C. Therefore, this preserves the property that for $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^C - E^C$,

$$\forall \bar{\mathsf{v}}.\ Q_\ell(\bar{\mathsf{v}}) \rightarrow \neg\mathcal{R}(\bar{\mathsf{v}}).$$

Now, if the **while** loop terminates, it must be the case that $E^C = \emptyset$. Thus, on exit of the loop, condition (7) is fulfilled.

Finally, if Non-Terminating is returned, then there is a location $\ell \in \mathcal{L}^C$ and a state satisfying $\sigma \models Q_\ell(\overline{\mathsf{v}})$ such that configuration (ℓ, σ) is reachable. That is, condition (5) is satisfied.

Hence, all conditions of Theorem 1 are fulfilled. Therefore, P does not terminate. Moreover, the proof of Theorem 1 gives a constructive way of building an infinite computation by means of $\mathcal{Q}$, $\mathcal{U}$ and π. $\square$

Note that constraint (9):

$$\text{For } \ell \in \mathcal{L}^C : \exists \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \bigvee_{\tau=(\ell,\ell',\mathcal{R})\in\mathcal{T}^C} \mathcal{R}(\overline{\mathsf{v}})$$

is not actually used in the proof of Theorem 2, and thus is not needed for the correctness of the approach. Its purpose is rather to help Prove-NT to avoid getting into dead-ends unnecessarily. Namely, without (9) it could be the case that for some location $\ell \in \mathcal{L}^C$, we computed a quasi-invariant that forbids all transitions $\tau \in \mathcal{T}^C$ from ℓ. Since Prove-NT only strengthens quasi-invariants and does not backtrack, if this situation were reached the procedure would probably not succeed in proving non-termination.

Now let us describe how constraints are effectively solved. First of all, constraints (8), (9), and (11) are universally quantified over integer variables. Following the same ideas of constraint-based linear invariant generation [25], these constraints are soundly transformed into an existentially quantified formula in NRA by abstracting program and non-deterministic variables and considering them as reals, and then applying Farkas' Lemma [28]. As regards constraint (10), the alternation of quantifiers in

$$\forall \overline{\mathsf{v}} \exists \overline{\mathsf{u}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow U_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}}) \wedge N_\tau(\overline{\mathsf{v}}, \overline{\mathsf{u}})$$

is dealt with by introducing a template $P_{u,\tau}(\overline{\mathsf{v}}) \equiv p_{u,\tau,0} + \sum_{v\in\overline{\mathsf{v}}} p_{u,\tau,v} \cdot v$ for each $u \in \overline{\mathsf{u}}$ and skolemizing. This yields[6] the formula

$$\forall \overline{\mathsf{v}}.\ Q_\ell(\overline{\mathsf{v}}) \wedge M_\ell(\overline{\mathsf{v}}) \wedge \mathcal{R}(\overline{\mathsf{v}}) \rightarrow U_\tau(\overline{\mathsf{v}}, P_{\overline{\mathsf{u}},\tau}(\overline{\mathsf{v}})) \wedge N_\tau(\overline{\mathsf{v}}, P_{\overline{\mathsf{u}},\tau}(\overline{\mathsf{v}})),$$

which implies constraint (10), and to which the above transformation into NRA can be applied. Note that, since the Skolem function is not symbolic but an explicit linear function of the program variables, potentially one might lose solutions.

Finally, once a weighted formula in NRA containing hard and soft clauses is obtained, (some of the) existentially quantified variables are forced to take integer values, and the resulting problem is handled by a Max-SMT(NIA) solver [27, 29]. In particular, the unknowns of the templates $P_{u,\tau}(\overline{\mathsf{v}})$ introduced for skolemizing non-deterministic variables are imposed to be integers. Since program variables have integer type, this guarantees that only integer values are assigned in the non-deterministic assignments of the infinite computation that proves non-termination.

There are some other issues about our implementation of the procedure that are worth mentioning. Regarding how the weights of the soft clauses are determined, we follow a heuristic aimed at discarding "difficult" transitions in $\mathcal{E}^C$ as soon as possible. Namely, the edge-closing constraint (11) of transition $\tau = (\ell, \ell', \mathcal{R}) \in \mathcal{E}^C$ is given a

[6] Again, existential quantifiers over template unknowns are implicit.

weight which is inversely proportional to the number of literals in $\mathcal{R}(\overline{v})$. Thus, transitions with few literals in their conditional part are associated with large weights, and therefore the Max-SMT solver prefers to discard them over others. The rationale is that for these transitions there may be more states that satisfy the conditional part, and hence they may be more difficult to rule out. Altogether, it is convenient to get rid of them before quasi-invariants become too constrained.

Finally, as regards condition (3), our implementation can actually handle transition systems for which this condition does not hold. This may be interesting in situations where, e.g., non-determinism is present in conditional statements, and one does not want to introduce additional variables and locations as was done in Section 2.2 for presentation purposes. The only implication of overriding condition (3) is that, in this case, the properties that must be discarded in soft clauses of condition (11) are not the transitions leaving the SCSG under consideration, but rather the negation of the transitions staying within the SCSG.

5 Experiments

In this section we evaluate the performance of a prototype implementation of the techniques proposed here in our termination analyzer CppInv, available at `www.lsi.upc.edu/~albert/cppinv-CAV.tar.gz` together with all of the benchmarks. This tool admits code written in (a subset of) C++ as well as in the language of T2 [20]. The system analyses programs with integer variables, linear expressions and function calls, as well as array accesses to some extent. As a reachability checker we use CPA [30].

Altogether, we compare CppInv with the following tools:

- T2 [20] version CAV'13 (henceforth, T2-CAV), which implements an algorithm that tightly integrates invariant generation and termination analysis [19].
- T2 [20] version TACAS'14 (henceforth, T2-TACAS), which reduces the problem of proving non-termination to the search of an under-approximation of the program guided by a safety prover [31].
- Julia [32], which implements a technique described by Payet and Spoto [33] that reduces non-termination to constraint logic programming.
- AProVE [11] with the Java Bytecode front-end, which uses the SMT-based non-termination analysis proposed in [34].
- A reimplementation of TNT [35] by the authors of [31] that uses Z3 [36] as an SMT back-end.

Unfortunately, because of the unavailability of some of the tools (T2-TACAS, T2-CAV, TNT) or the fact that they do not admit a common input language (Julia, AProVE), it was not possible to run all these systems on the same benchmarks on the same computer. For this reason, for each of the tables below we consider a different family of benchmarks taken from the literature and provide the results of executing our tool (on a 3.40 GHz Intel Core i7 with 16 GB of RAM) together with the data of competing systems reported in the respective publications. Note that the results of third-party systems in those publications may have some inaccuracies, due to, e.g., the conversion of benchmarks in different formats. However, in those cases the distances between the tools seem to be significant enough to draw conclusions on their relative performance.

Table 1 shows comparative results on benchmarks taken from [31]. In that paper, the tools T2-TACAS, AProVE, Julia and TNT are considered. The time limit is set to 60 seconds both in that work as well as in the executions of CppInv. The benchmarks are classified according to three categories: (a) all the examples in the benchmark suite known to be non-terminating previously to [31]; (b) all the examples in the benchmark suite known to be terminating previously to [31]; and (c) the rest of instances. Rows of the table correspond to non-termination provers. Columns are associated to each of these three categories of problems. Each column is split into three subcolumns reporting the number on "non-terminating" answers, the number of timed outs, and the number of other answers (which includes "terminating" and "unknown" answers), respectively. Even with the consideration that experiments were conducted on different machines, the results in columns (a) and (c) of Table 1 show the power of the proposed approach on these examples. As for column (b), we found out that instance `430.t2` was wrongly classified as terminating. Our witness of non-termination has been manually verified.

Table 1. Experiments with benchmarks from [31]

	(a)			(b)			(c)		
	Nonterm	TO	Other	Nonterm	TO	Other	Nonterm	TO	Other
CppInv	70	6	5	1	16	237	113	35	9
T2-TACAS	51	0	30	0	45	209	82	3	72
AProVE	0	61	20	0	142	112	0	139	18
Julia	3	8	70	0	40	214	0	91	66
TNT	19	3	59	0	48	206	32	12	113

Table 2 (a), which follows a similar format to Table 1, compares CppInv, T2-CAV and AProVE on benchmarks from [19] (all with a time limit of 300 seconds). Note that, in the results reported in [19], due to a wrong abstraction in the presence of division, T2 was giving two wrong non-termination answers (namely, for the instances `rlft3.t2` and `rlft3.c.i.rlft3.pl.t2.fixed.t2`, for which the termination proofs produced by CppInv[24] have been checked by hand). For this reason we have discarded those two programs from the benchmark suite. In this case, the performance of our tool is slightly worse than that of T2-CAV. However, it has to be taken into account that T2-CAV was exploiting the cooperation between the termination and the non-termination provers, while we still do not apply this kind of optimizations.

In Table 2 (b), CppInv is compared with the results of Julia and AProVE from [34] on Java programs coming from [37]. CppInv was run on C++ versions of these benchmarks, which admitted a direct translation from Java. The time limit was set to 60 seconds. Columns represent respectively the number of terminating instances (YES), non-terminating instances (NO), instances for which the construction of the proof failed before the time limit (MAYBE), and timeouts (TO). For these instances AProVE gets slightly better results than CppInv. However, it should be taken into account that four programs of this set of benchmarks include non-linear expressions, which we cannot handle. Moreover, when compared on third-party benchmarks (see Tables 1 and 2 (a)), our results are better.

Finally, Table 2 (c) shows the results of running our tool on programs from the on-line programming learning environment Jutge.org [38] (see `www.jutge.org`), which is currently being used in several programming courses in the Universitat Politècnica de Catalunya. As a paradigmatic example in which it is easy to write wrong non-terminating code, we have considered the exercise **Binary Search**. The programs in this benchmark suite can be considered challenging since, having been written by students, their structure is often more complicated than necessary. In this case the time limit was 60 seconds. As can be seen from the results, for a ratio of 89% of the cases, CppInv is able to provably determine in less than one minute if the program is terminating or not.

Table 2. Experiments with benchmarks from [19] (a), from [37] (b) and from Jutge.org (c)

(a)

	Nonterm	TO	Other
CppInv	167	39	243
T2-CAV	172	14	263
AProVE	0	51	398

(b)

	YES	NO	MAYBE	TO
CppInv	1	44	9	1
AProVE	1	51	0	3
Julia	1	0	54	0

(c)

	YES	NO	MAYBE	TO
Binary search	2745	484	22	391

All in all, the experimental results show that our technique, although it is general and is not tuned to particular problems, is competitive with the state of the art and performs reasonably and uniformly well on a wide variety of benchmarks.

6 Related work

Several systems have been developed in recent years for proving non-termination. One of these is, e.g., the tool TNT [35], which proceeds in two phases. The first phase exhaustively generates candidate lassos. The second one checks each lasso for possible non-termination by seeking a *recurrent set* of states, i.e., a set of states that is visited infinitely often along the infinite path that results from unrolling the lasso. This is carried out by means of constraint solving, as in our approach. But while there is an infinite number of lassos in a program, our SCSGs can be finitely enumerated. Further, we can handle unbounded non-determinism, whereas TNT is limited to deterministic programs.

Other methods for proving non-termination that use an off-the-shelf reachability checker like our technique have also been proposed [39, 31]. In [39], the reachability checker is used on instrumented code for inferring weakest preconditions, which give the most general characterization of the inputs under which the original program is non-terminating. While in [39] non-determinism can be dealt with in a very restricted manner, the method in [31] can deal with unbounded non-determinism as we do. In the case of [31], the reachability checker is iteratively called to eliminate every terminating path through a loop by restricting the state space, and thus may diverge on many loops. Our method does not suffer from this kind of drawbacks.

Some other approaches exploit theorem-proving techniques. For instance, the tool INVEL [37] analyzes non-termination of Java programs using a combination of theorem proving and invariant generation. INVEL is only applicable to deterministic programs. Another tool for proving non-termination of Java programs is APROVE [11], which uses SMT solving as an underlying reasoning engine. The main drawback of their method is that it is required that either recurrent sets are singletons (after program slicing) or loop conditions themselves are invariants. Our technique does not have such restrictions.

Finally, the tool TREX [40] integrates existing non-termination proving approaches within a TERMINATOR-like [41] iterative procedure. Unlike TREX, which is aimed at sequential code, Atig et al. [42] focus on concurrent programs: they describe a non-termination proving technique for multi-threaded programs, via a reduction to non-termination reasoning for sequential programs. Our work should complement both of these approaches, since we provide significant advantages over the underlying non-termination proving tools that were previously used.

7 Conclusions and Future Work

In this paper we have presented a novel Max-SMT-based technique for proving that programs do not terminate. The key notion of the approach is that of a *quasi-invariant*, which is a property such that if it holds at a location during execution once, then it continues to hold at that location from then onwards. The method considers an SCSG of the control flow graph at a time, and thanks to Max-SMT solving generates a quasi-invariant for each location. Weights of soft constraints guide the solver towards quasi-invariants that are also *edge-closing*, i.e., that forbid any transition exiting the SCSG. If an SCSG with edge-closing quasi-invariants is reachable, then the program is non-terminating. This last check is performed with an off-the-shelf reachability checker. We have reported experiments with encouraging results that show that a prototypical implementation of the proposed approach has comparable and often better efficacy than state-of-the-art non-termination provers.

As regards future research, a pending improvement is to couple the reachability checker with the quasi-invariant generator, so that the invariants synthesized by the former in unsuccessful attempts are reused by the latter when producing quasi-invariants. Another line for future work is to combine our termination [24] and non-termination techniques. Following a similar approach to [19], if the termination analyzer fails, it can communicate to the non-termination tool the transitions that were proved not to belong to any infinite computation. Conversely, when a failed non-termination analysis ends with an unsuccessful reachability check, one can pass the computed invariants to the termination system, as done in [40]. Finally, we also plan to extend our programming model to handle more general programs (procedure calls, non-linearities, etc.).

Acknowledgments. We thank the Jutge.org team for providing us with benchmarks.

References

1. Francez, N., Grumberg, O., Katz, S., Pnueli, A.: Proving Termination of Prolog Programs. In Parikh, R., ed.: Logic of Programs. Volume 193 of Lecture Notes in Computer Science., Springer (1985) 89–105
2. Dams, D., Gerth, R., Grumberg, O.: A heuristic for the automatic generation of ranking functions. In: Workshop on Advances in Verification. (2000) 1–8
3. Colón, M., Sipma, H.: Practical methods for proving program termination. In Brinksma, E., Larsen, K.G., eds.: Proc. CAV '02. Volume 2404 of LNCS., Springer (2002) 442–454
4. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In Steffen, B., Levi, G., eds.: Proc. VMCAI '04. Volume 2937 of LNCS., Venice, Italy, Springer (2004) 239–251
5. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of polynomial programs. In Cousot, R., ed.: Proc. VMCAI '05. Volume 3385 of LNCS., Springer (2005) 113–129
6. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Proc. ICALP '05. Volume 3580 of LNCS., Springer (2005) 1349–1361
7. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In Etessami, K., Rajamani, S.K., eds.: Proc. CAV '05. Volume 3576 of LNCS., Springer (2005) 491–504
8. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination analysis of integer linear loops. In Abadi, M., de Alfaro, L., eds.: CONCUR. Volume 3653 of Lecture Notes in Computer Science., Springer (2005) 488–502
9. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In Cousot, R., ed.: VMCAI. Volume 3385 of Lecture Notes in Computer Science., Springer (2005) 1–24
10. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. PLDI '06, ACM Press (2006) 415–426
11. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Proc. IJCAR '06. Volume 4130 of LNAI., Springer (2006) 281–286
12. Podelski, A., Rybalchenko, A.: ARMC: the logical choice for software model checking with abstraction refinement. In: Proc. PADL '07. Volume 4354 of LNCS., Springer (2007) 245–259
13. Babic, D., Hu, A.J., Rakamaric, Z., Cook, B.: Proving termination by divergence. In: SEFM, IEEE Computer Society (2007) 93–102
14. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In Gupta, A., Malik, S., eds.: Proc. CAV '08. Volume 5123 of LNCS., Springer (2008) 328–340
15. Cook, B., Podelski, A., Rybalchenko, A.: Summarization for termination: no return! Formal Methods in System Design **35**(3) (2009) 369–387
16. Otto, C., Brockschmidt, M., Essen, C.v., Giesl, J.: Automated termination analysis of Java Bytecode by term rewriting. In: Proc. RTA '10. Volume 6 of LIPIcs., Edinburgh, UK, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010) 259–276
17. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.: Termination analysis with compositional transition invariants. In: Proc. CAV '10. Volume 6174 of LNCS., Springer (2010) 89–103
18. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop summarization and termination analysis. In: Proc. TACAS '11. Volume 6605 of LNCS., Springer (2011) 81–95
19. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Proc. CAV '13. LNCS (2013)

20. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Proc. TACAS '13. Volume 7795 of LNCS., Springer (2013) 47–61
21. Babic, D., Cook, B., Hu, A.J., Rakamaric, Z.: Proving termination of nonlinear command sequences. Formal Asp. Comput. **25**(3) (2013) 389–403
22. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. Formal Methods in System Design **43**(1) (2013) 93–120
23. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: Smt-based array invariant generation. In Giacobazzi, R., Berdine, J., Mastroeni, I., eds.: VMCAI. Volume 7737 of Lecture Notes in Computer Science., Springer (2013) 169–188
24. Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving Termination of Imperative Programs Using Max-SMT. In: Proc. FMCAD '13. (2013)
25. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Proc. CAV '03. (2003) 420–432
26. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (February 2009)
27. Nieuwenhuis, R., Oliveras, A.: On SAT Modulo Theories and Optimization Problems. In Biere, A., Gomes, C.P., eds.: SAT. Volume 4121 of Lecture Notes in Computer Science., Springer (2006) 156–169
28. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (June 1998)
29. Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Minimal-Model-Guided Approaches to Solving Polynomial Constraints and Extensions. Submitted (2014)
30. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In Gopalakrishnan, G., Qadeer, S., eds.: CAV. Volume 6806 of Lecture Notes in Computer Science., Springer (2011) 184–190
31. Chen, H.Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Proc. TACAS '14. To appear. (2014)
32. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for Java bytecode based on path-length. ACM TOPLAS **32**(3) (2010)
33. Payet, É., Spoto, F.: Experiments with Non-Termination Analysis for Java Bytecode. Electr. Notes Theor. Comput. Sci. **253**(5) (2009) 83–96
34. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and `NullPointerException`s for Java Bytecode. In: Proc. FoVeOOS '11. Volume 7421 of LNCS., Turin, Italy, Springer (2012) 123–141
35. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In Necula, G.C., Wadler, P., eds.: Proc. POPL '08, ACM Press (2008) 147–158
36. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In Ramakrishnan, C.R., Rehof, J., eds.: Proc. TACAS '08. Volume 4963 of LNCS., Springer (2008) 337–340
37. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In Beckert, B., Hähnle, R., eds.: Proc. TAP '08. Volume 4966 of LNCS., Springer (2008) 154–170
38. Petit, J., Giménez, O., Roura, S.: Jutge.org: an educational programming judge. In King, L.A.S., Musicant, D.R., Camp, T., Tymann, P.T., eds.: SIGCSE, ACM (2012) 445–450
39. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In Gupta, R., Amarasinghe, S.P., eds.: Proc. PLDI '08, ACM Press (2008) 281–292
40. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for termination. In: Proc. SAS '10. Volume 6337 of LNCS., Springer (2010) 304–319
41. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In Ball, T., Jones, R.B., eds.: Proc. CAV '06. Volume 4144 of LNCS., Seattle, WA, USA, Springer (2006) 415–418
42. Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multi-threaded programs. In Madhusudan, P., Seshia, S.A., eds.: Proc. CAV '12. Volume 7358 of LNCS., Springer (2012) 210–226