Proving Termination of Imperative Programs Using Max-SMT

Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, Albert Rubio
Universitat Politècnica de Catalunya, Barcelona, Spain

***Abstract*—We show how Max-SMT can be exploited in constraint-based program termination proving. Thanks to expressing the generation of a ranking function as a Max-SMT optimization problem where constraints are assigned different weights, quasi-ranking functions –functions that almost satisfy all conditions for ensuring well-foundedness– are produced in a lack of ranking functions. By means of trace partitioning, this allows our method to progress in the termination analysis where other approaches would get stuck. Moreover, Max-SMT makes it easy to combine the process of building the termination argument with the usually necessary task of generating supporting invariants. The method has been implemented in a prototype that has successfully been tested on a wide set of programs.**

I. Introduction

Proving termination is necessary to ensure total correctness of programs. Still, termination bugs are difficult to trace and are hardly notified: as they do not arise as system failures but as unresponsive behavior, when faced to them users tend to restart their devices without reporting to software developers. Due to this, approaches for proving termination of imperative programs have regained an increasing interest in the last decade [1]–[4].

One of the major difficulties in these methods is that often *supporting invariants* are needed. E.g., in [5] linear invariants are exhaustively computed before termination analysis. In the same paper a heuristic approach is also presented, which only requires a light-weight invariant generator by restricting to single-variable ranking functions. Another solution is proposed in [6], where invariant generation is not performed eagerly but on demand. By formulating both invariant and ranking function synthesis as constraint problems, both can be solved simultaneously, so that only the necessary supporting invariants for the targeted ranking functions –namely, *lexicographic linear ranking functions*– need to be discovered.

Based on [5], [6], we present a Max-SMT constraint-based approach for proving termination. The crucial observation in our method is that, albeit our goal is to show that transitions cannot be executed infinitely by finding a ranking function or an invariant that disables them, if we only discover an invariant, or an invariant and a *quasi-ranking function* that almost fulfills all needed properties for well-foundedness, we have made some progress: either we can remove *part of a transition* and/or we have improved our knowledge on the behavior of the program. A natural way to implement this idea is by considering that some of the constraints are *hard* (the ones guaranteeing invariance) and others are *soft* (those guaranteeing well-foundedness) in a Max-SMT framework. Moreover, by giving different weights to the constraints we can set priorities and favor those invariants and (quasi-) ranking functions that lead to the furthest progress.

The technique has been implemented in our tool CppInv, which analyses programs with integer variables and linear expressions. Thanks to it, we have proved termination of a wide set of programs, which have been taken from the programming learning environment Jutge.org [7] and from benchmark suites in the literature [8].

A. Related Work.

As mentioned above, our method is based on [5]. Namely, we have borrowed the core argument for termination proofs, which is based on iteratively discarding those transitions that cannot be executed infinitely. However, we improve on the way supporting invariants are generated. While in [5] invariants are pre-computed in a process that is independent from the termination analysis and which turns out to be the bottleneck of the approach, we find lazily the invariants needed to ensure that ranking functions meet their requirements.

Our research also builds upon [6], where the constraint-based method [9] was first applied to termination. However, we extend this work in several aspects. First, in that approach only linear programs with unnested loops can be handled, while we can deal with arbitrary control structures. Moreover, in [6] the generation of their lexicographic ranking functions requires a higher-level loop that, before sending the constraint problem to the solver, determines the precedence of the transitions in the lexicographic order. On the other hand, in our approach this outer loop is not needed. Finally, thanks to assigning weights to the constraints, unlike [6] we do not need to stipulate the number of supporting invariants that will be needed a priori, and hence our constraint problems are simpler. Further, weights allow us to guide the solving engine in the search of appropriate ranking functions and invariants.

In [10], the lexicographic approach of [6] is extended so as to handle programs with complex control flow. However, their method still requires to search for the right ordering of the transitions in order to obtain a successful termination proof. Moreover, in this technique the procedures for synthesizing ranking functions and auxiliary invariants do not share enough information, while in our proposal these mechanisms are tightly coupled. Finally, in [8] a method closely related to ours is presented. Both approaches, which have been developed independently, go in the same direction of achieving a better cooperation between the invariant and the ranking function

syntheses. Still, a significant difference is that we can exploit the quasi-ranking functions produced in the absence of ranking functions in order to progress in the termination analysis.

In addition to lexicographic ranking functions, there is a group of effective tools whose termination arguments are based on Ramsey's Theorem and the notion of *transition invariant* [11]. Transition invariants are over-approximations of the transitive closure of the transition relation restricted to the reachable state space. The crucial observation is that a transitive relation that is *disjunctively well-founded*, i.e., that is included in the union of well-founded relations, must be well-founded too. Hence, if one is able to find a transition invariant that is also disjunctively well-founded, the program must be terminating. In [12], this transition invariant is computed iteratively, starting from the empty relation, by discovering unranked paths of the program thanks to a reachability check, and using the approach in [3] for synthesizing new ranking functions for them. On the other hand, in [13] the generation of the disjunctively well-founded transition invariant is performed bottom-up from innermost loops by identifying invariant and transitive relations among a set of templates that are disjunctively well-founded by construction. Nested loops are then handled thanks to loop summarization. Our techniques can also be seen as producing a disjunctively well-founded transition invariant, being the difference with respect to the previous approaches in the way new unranked paths are identified and how a termination argument is generated for them.

Finally, a problem related to proving termination that has recently raised interest in the area is that of *conditional termination:* to synthesize automatically preconditions on the inputs that ensure program termination. In this context, in [15] the authors consider what they call *potential ranking functions*, which are functions over program states that are bounded but not necessarily decreasing. The quasi-ranking functions that we consider here are more general, as for instance functions that are decreasing but not bounded are also included. In [16], the problem of conditional termination is also considered. The approach is based on disjunctively well-founded relations as in [12], but instead of identifying unranked program paths, thanks to a dual inclusion the authors partition the transition relation into those behaviors already proved to be terminating and those whose status is still unknown. In our work we also proceed by splitting the transition relation into a terminating part and an unknown part. However, in [16] this division is achieved by means of a fixpoint computation, while our approach is constraint-based.

II. Preliminaries

A. 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, to determine whether

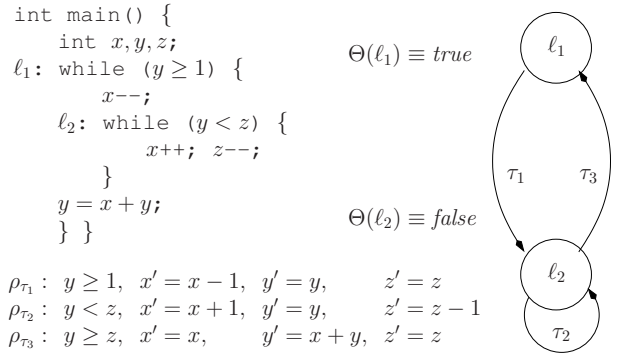


Fig. 1. Program and its transition system.

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 [17]: to decide the satisfiability of a given quantifier-free first-order formula with respect to a background theory. Here we will consider the theories of *linear arithmetic (LA)*, where literals are linear inequalities, and the more general theory of *non-linear arithmetic (NA)*, where literals are polynomial inequalities.

Another generalization of SAT is the *Max-SAT* problem [17]: it consists in, given a *weighted* formula F where each clause C_i has a weight ω_i (a positive number or infinity), to find 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*[18] 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.

B. Transition Systems, Invariants and Ranking Functions

Henceforth we will model imperative programs by means of *transition systems*. A transition system $\mathcal{S} = (\overline{v}, \mathcal{L}, \Theta, \mathcal{T})$ consists of a tuple of *variables* $\overline{v}$, 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 (ℓ, ℓ', ρ), where ℓ, $\ell' \in \mathcal{L}$ are the *pre* and *post* locations respectively, and ρ is the *transition relation*: a formula over the program variables $\overline{v}$ and their primed versions $\overline{v}'$, which represent the values of the variables after the transition. See Fig. 1 for an example of a program together with a corresponding representation as a transition system.

From now on we assume that variables take *integer* values and programs are *linear*, i.e., the initial conditions Θ and transition relations ρ are described as conjunctions of linear inequalities. Strict inequalities may be translated into non-strict ones thanks to the integer type of the variables.

A *state* is an assignment of a value to each of the variables in $\overline{v}$. A *configuration* is a pair (ℓ, σ) consisting of a location ℓ and a state σ. A *computation* is a (possibly infinite) 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 (ℓ_i, σ_i) and $(\ell_{i+1}, \sigma_{i+1})$, there exists a transition $\tau = (\ell_i, \ell_{i+1}, \rho) \in \mathcal{T}$ such that $(\sigma_i, \sigma_{i+1}) \models \rho$. A configuration (ℓ, σ) is *reachable* if there exists a computation ending at (ℓ, σ). A transition system is said to be *terminating* if all its computations are finite. The problem that we target in this work is, given a transition system, to determine if it is terminating or not.

A transition $\tau = (\ell, \ell', \rho)$ is *disabled* if it can never be executed, i.e., if for all reachable configuration (ℓ, σ), there does not exist any σ' such that $(\sigma, \sigma') \models \rho$. A transition τ is called *finitely executable* if in any computation, τ is only executed a finite number of times (in particular, if τ is disabled). Otherwise, i.e., if there exists a computation where τ is executed infinitely, we say that τ is *infinitely executable*.

An *assertion* is a first-order formula over $\overline{v}$. An assertion I is an *invariant* at location ℓ if for any reachable configuration (ℓ, σ), it holds that $\sigma \models I$. An *invariant map* μ assigns an invariant $\mu(\ell)$ to each of the locations ℓ. An important class of invariant maps is that of *inductive invariant maps*:

Definition 1: An invariant map μ is said to be *inductive* if:

- **[Initiation]** For every location $\ell \in \mathcal{L}$: $\Theta(\ell) \models \mu(\ell)$
- **[Consecution]** For every transition $\tau = (\ell, \ell', \rho) \in \mathcal{T}$: $\mu(\ell) \wedge \rho \models \mu(\ell')'$.

Invariant maps are fundamental when analyzing program termination. For instance, a transition $\tau = (\ell, \ell', \rho)$ is proved to be disabled if there is an invariant $\mu(\ell)$ at location ℓ such that $\mu(\ell) \wedge \rho$ is unsatisfiable. In general, if μ is an invariant map, then any transition $\tau = (\ell, \ell', \rho)$ can be safely strengthened by replacing the transition relation ρ by $\mu(\ell) \wedge \rho$.

The basic idea of the approach we follow for proving program termination [5] is to argue by contradiction that no transition is infinitely executable. First of all, no disabled transition can be infinitely executable trivially. Moreover, one just needs to focus on transitions joining locations in the same strongly connected component (SCC): if a transition is executed over and over again, then its pre and post locations must belong to the same SCC. So let us assume that one has found a *ranking function* for such a transition τ, according to:

Definition 2: Let $\tau = (\ell, \ell', \rho)$ be a transition such that ℓ and ℓ' belong to the same SCC, denoted by C. A function $R : \overline{v} \to \mathbb{Z}$ is said to be a *ranking function* for τ if:

- **[Boundedness]** $\rho \models R \geq 0$
- **[Strict Decrease]** $\rho \models R > R'$
- **[Non-increase]** For every $\hat{\tau} = (\hat{\ell}, \hat{\ell}', \hat{\rho}) \in \mathcal{T}$ such that $\hat{\ell}, \hat{\ell}' \in C$: $\hat{\rho} \models R \geq R'$

Note that boundedness and strict decrease *only* depend on τ, while non-increase depends on *all* transitions in the SCC.

The key result is that if $\tau = (\ell, \ell', \rho)$ admits a ranking function R, then it is finitely executable. Indeed, first notice that if one can execute τ from a configuration (ℓ, σ) then $R(\sigma) \geq 0$, because of boundedness. Also, the value of R at the states along any path contained in C cannot increase, thanks to the non-increase property. Moreover, in any cycle contained in C traversing τ, the value of R strictly decreases, due to the strict decrease property. Now, let us assume that there was a computation where τ was executed infinitely. Such a computation would eventually visit only locations in C. Because of the previous observations, by evaluating R at the states at which τ is executed we could construct an infinitely decreasing sequence of non-negative integers, a contradiction.

Finitely executable transitions can be safely removed from the transition system as regards termination analysis. This in turn may break the SCC's into smaller pieces. If by applying this reasoning recursively one can prove that all transitions are finitely executable, then the transition system is terminating.

C. Constraint-Based Program Analysis

Here we review the *constraint-based program analysis* approach [6], [9]. The idea is to consider a template for candidate invariant properties (respectively, ranking functions), e.g., linear inequalities (linear expressions). These templates involve both program variables as well as unknowns whose values have to be determined so as to ensure the required properties. To this end, the implications in Definition 1 (Definition 2) are expressed by means of *constraints* (hence the name of the approach) on the unknowns. If implications are encoded soundly, any solution to the constraints yields an invariant map (ranking function). Specifically, if linear arithmetic is the target language, this can be achieved with Farkas' Lemma:

Theorem 1 (Farkas' Lemma): Let S be a system of linear inequalities $Ax + b \leq 0$ ($A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$) over real variables $x^T = (x_1, \ldots, x_n)$. When S is satisfiable, it entails a linear inequality $c^T x + d \leq 0$ ($c \in \mathbb{R}^n, d \in \mathbb{R}$) iff there is $\lambda \in \mathbb{R}^m$ such that $\lambda \geq 0$, $c^T = \lambda^T A$ and $d \leq \lambda^T b$. Further, S is unsatisfiable iff $1 \leq 0$ can be so derived.

For clarity, henceforth the following notation is used. Given a conjunction of linear inequalities $Ax + b \leq 0$ and a linear inequality $c^T x + d \leq 0$, where the coefficients a_{ij}, b_i, c_j, d may be real numbers or unknowns, we denote by $Ax + b \leq 0 \vdash c^T x + d \leq 0$ the set of constraints on the unknown coefficients and on fresh real unknowns $\lambda = (\lambda_1, \ldots, \lambda_m)$, consisting in $\lambda \geq 0$, $c^T = \lambda^T A$ and $d \leq \lambda^T b$.

III. Termination Analysis with Max-SMT

In this section we first describe a constraint-based method for termination analysis that uses SMT and identify some of its shortcomings (Sect. III-A). Then we show how Max-SMT can be used to overcome these limitations (Sect. III-B).

A. An SMT Approach to Proving Termination

Following the approach described in Sect. II-B [5], to show that a transition τ is finitely executable and thus discard it, one needs either a disability argument or a ranking function for it. To this end we construct a constraint system, i.e. an SMT formula, whose solutions correspond to either an invariant that proves disability, or a ranking function. Given an SCC, the constraint system, if satisfiable, will allow discarding (at least,

but possibly more than) one of the transitions in the SCC. By iterating this procedure until no cycles are left we will obtain a termination argument for the SCC.

To construct the constraint system, first of all we consider:

- for each location ℓ, a linear invariant template $I_\ell(\overline{v}) \equiv i_{\ell,0} + \sum_{v\in\overline{v}} i_{\ell,v}\cdot v \leq 0$, where $i_{\ell,0}$, $i_{\ell,v}$ are unknown;
- a linear ranking function template $R(\overline{v}) \equiv r_0 + \sum_{v\in\overline{v}} r_v\cdot v$, where r_0, r_v are unknown.

Recall that ranking functions are associated to transitions, not to locations. However, instead of introducing a template for each transition, we just have one single template, which, if the constraint system has a solution, will be a ranking function for a transition *to be determined by the solver*.

Similarly to [6], we take the following constraints from the definitions of inductive invariant and ranking function:

Initiation:	For $\ell \in \mathcal{L}$:	$\mathbb{I}_\ell \stackrel{def}{=} \Theta(\ell) \vdash I_\ell$
Disability:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{D}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash 1 \leq 0$
Consecution:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{C}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash I'_{\ell'}$
Boundedness:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{B}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq 0$
Strict Decrease:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{S}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R > R'$
Non-increase:	For $\tau = (\ell,\ell',\rho) \in \mathcal{T}$:	$\mathbb{N}_\tau \stackrel{def}{=} I_\ell \wedge \rho \vdash R \geq R'$

Let L and T be the sets of locations and transitions in the SCC in hand, respectively. Let also P be the set of *p*ending transitions, i.e., which have not been proved to be finitely executable yet. Then we build the next constraint system:

$$\bigwedge_{\ell\in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau\in T} (\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge \bigvee_{\tau\in P} (\mathbb{D}_\tau \vee (\mathbb{B}_\tau \wedge \mathbb{S}_\tau)) \wedge ((\bigwedge_{\tau\in P} \mathbb{N}_\tau) \vee \bigvee_{\tau\in P} \mathbb{D}_\tau).$$

The first two conjuncts guarantee that an invariant map is computed; the other two, that at least one of the pending transitions can be discarded. Notice that, if there is no disabled transition, we ask that *all* transitions in P are non-increasing, but only that at least *one* transition in P (the next to be removed) is both bounded and strict decreasing. Note also that for finding invariants one has to take into account *all* transitions in the SCC, even those that have already been proved to be finitely executable: otherwise some reachable states might not be covered, and the invariant generation would become unsound. Hence in our termination analysis we consider two transition systems: the original transition system for invariant synthesis, whose transitions are T and which remains all the time the same; and the *termination transition system*, whose transitions are P, i.e, where transitions already shown to be finitely executable have been removed. This duplication is similar to the *cooperation graph* of [8].

However, this first approach is problematic when a ranking function needs several invariants. A possible solution is to add more templates iteratively, so that for example initially invariants consisting of a single linear inequality are tried, if unsuccessful then invariants consisting of a conjunction of two linear inequalities are tried, etc. But when proceeding in this way, all problems before the right number of invariants is found are unsatisfiable. This is wasteful, as no constructive information is retrieved from unsatisfiable constraint systems.

Another problem with this method for analyzing termination is that the kind of termination proofs it yields may be too restricted. More specifically, when one proves that a transition τ is finitely executable, then a single termination argument shows there is no computation where τ appears infinitely. Although this produces compact proofs, on the other hand sometimes there may not exist such a unique reason for termination, and it becomes necessary a more fine-grained examination. However, the approach as presented so far does not provide a natural way or guidance for refining the analysis.

B. A Max-SMT Approach to Proving Termination

The main contribution of our work is to show that the constraint system can be expressed in such a way that, even when it turns out to be unsatisfiable, some information useful for refining the termination analysis can be obtained. The key observation is that, even though our aim is to prove transitions to be finitely executable (by finding a ranking function or an invariant that disables them), if we just find an invariant, or an invariant and a *quasi-ranking function* that is close to fulfill all required conditions, we have progressed in our analysis.

The idea is to consider the constraints guaranteeing invariance as *hard*, so that any solution to the constraint system will satisfy them, while the rest are *soft*. Let us consider propositional variables $p_\mathbb{B}$, $p_\mathbb{S}$ and $p_\mathbb{N}$, which intuitively represent if the conditions of boundedness, strict decrease and non-increase in the definition of ranking function are violated respectively, and corresponding weights $\omega_\mathbb{B}$, $\omega_\mathbb{S}$ and $\omega_\mathbb{N}$. We consider now the next constraint system (where soft constraints are written $[\cdot,\omega]$, and hard ones as usual):

$$\bigwedge_{\ell\in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau\in T} \Big(\mathbb{D}_\tau \vee \mathbb{C}_\tau\Big) \wedge \bigvee_{\tau\in P} \Big(\mathbb{D}_\tau \vee \big((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S})\big)\Big) \wedge \Big((\bigwedge_{\tau\in P} \mathbb{N}_\tau) \vee \bigvee_{\tau\in P} \mathbb{D}_\tau \vee p_\mathbb{N}\Big) \wedge [\neg p_\mathbb{B}, \omega_\mathbb{B}] \wedge [\neg p_\mathbb{S}, \omega_\mathbb{S}] \wedge [\neg p_\mathbb{N}, \omega_\mathbb{N}].$$

Note that ranking functions have cost 0, and (if no transition is disabled) functions that fail in any of the conditions are penalized with the respective weight. Thus, the Max-SMT solver looks for the best solution and gets a ranking function if feasible; otherwise, the weights guide the search to get invariants and quasi-ranking functions that satisfy as many conditions as possible.

Hence this Max-SMT approach allows recovering information even from problems that would be unsatisfiable in the initial method. This information can be exploited to perform dynamic trace partitioning [19] as follows. Assume that the optimal solution to the above Max-SMT formula has been computed, and let us consider a transition $\tau \in P$ such that $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S}))$ evaluates to true in the solution. Then we distinguish several cases depending on the properties satisfied by τ and the computed function R:

- If τ is disabled then it can be removed.
- If R is non-increasing and satisfies boundedness and strict decrease for τ, then τ can be removed too: R is a ranking function for it.
- If R is non-increasing and satisfies boundedness for τ but not strict decrease, one can split τ in the termination

transition system into two new transitions: one where $R > R'$ is added to τ, and another one where $R = R'$ is enforced. Then the new transition with $R > R'$ is automatically eliminated, as R is a ranking function for it. Equivalently, this can be seen as adding $R = R'$ to τ. Now, if the solver could not prove R to be a true ranking function for τ because it was missing an invariant, this transformation will guide the solver to find that invariant so as to disable the transition with $R = R'$.

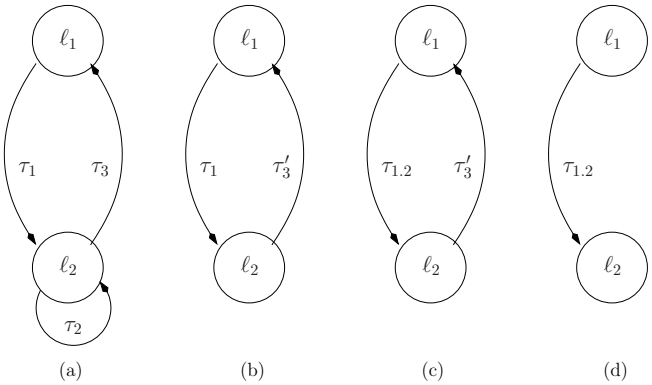
- If R is non-increasing and satisfies strict decrease for τ but not boundedness, the same technique from above can be applied: it boils down to adding $R < 0$ to τ.
- If R is non-increasing but neither strict decrease nor boundedness are fulfilled for τ, then τ can be split into two new transitions: one with $R < 0$, and another one with $R \geq 0 \wedge R = R'$.
- If R does not satisfy the non-increase property, then it is rejected; however, the invariant map from the solution can be used to strengthen the transition relations for the following iterations of the termination analysis.

Note this analysis may be worth applying on other transitions τ in the termination transition system apart from those that make $\mathbb{D}_\tau \vee ((\mathbb{B}_\tau \vee p_\mathbb{B}) \wedge (\mathbb{S}_\tau \vee p_\mathbb{S}))$ true. E.g., if R is a ranking function for a transition τ but fails to be so for another one τ' because strict decrease does not hold, then, according to the above discussion, τ' can be strengthened with $R = R'$.

On the other hand, working in this iterative way requires imposing additional constraints to avoid getting to a standstill. Namely, in the case where non-increase does not hold and so one would like to exploit the invariant, it is necessary to impose that the invariant is not redundant. More in detail, let us consider a fixed location ℓ, and let $I_\ell^{(1)}, \ldots, I_\ell^{(k)}$ be the previously computed invariants at location ℓ. Then I_ℓ, the invariant to be generated at ℓ, is redundant if it is implied by $I_\ell^{(1)}$, ..., $I_\ell^{(k)}$, i.e., if $\mathbb{E}_\ell \stackrel{def}{=} \forall \overline{v}\ (I_\ell^{(1)}(\overline{v}) \wedge \ldots \wedge I_\ell^{(k)}(\overline{v}) \rightarrow I_\ell(\overline{v}))$. So we impose $p_\mathbb{N} \rightarrow \neg \bigwedge_{\ell \in L} \mathbb{E}_\ell$ to ensure that violating non-increase leads to non-redundant invariants. Conditions are added similarly to avoid redundant quasi-ranking functions.

Another advantage of this Max-SMT approach is that by using different weights we can express priorities over conditions. Since, as explained above, violating the property of non-increase invalidates the computed function R, it is convenient to make $\omega_\mathbb{N}$ the largest weight. On the other hand, when non-increase and boundedness are fulfilled but not strict decrease an equality is added to the transition, whereas when non-increase and strict decrease are fulfilled but not boundedness just an inequality is added. As we prefer the former to the latter, in our implementation (see Sect. V) we set $\omega_\mathbb{B} > \omega_\mathbb{S}$.

A further improvement is the generation of *termination implications*. A termination implication at a location ℓ is an assertion $J(\overline{v})$ such that any transition in the *termination transition system* that leads into ℓ implies it, i.e., it holds that $\rho \models J(\overline{v}')$, where ρ is the relation of the transition. Thus, J will *eventually* hold when ℓ is reached (although, unlike ordinary invariants, may not initially be true; see Example 1 below). Hence, it can be propagated forward in the termination transition system to the transitions going out from ℓ. To produce termination implications, for each location ℓ a new linear inequality template $J_\ell(\overline{v})$ is introduced and the following constraint is imposed: $\bigwedge_{\tau=(\hat{\ell},\ell,\rho)\in P}(\mathbb{D}_\tau \vee I_{\hat{\ell}} \wedge \rho \vdash J'_\ell)$. Additional constraints are enforced to ensure that new termination implications are not redundant with the already computed invariants and termination implications.



$\Theta(\ell_1) \equiv \mathit{true} \qquad \Theta(\ell_2) \equiv \mathit{false}$

$$
\begin{array}{lllll}
\rho_{\tau_1}: & & y \geq 1,\ x' = x-1,\ y' = y, & & z' = z \\
\rho_{\tau_{1.2}}: & x < 0,\ & y \geq 1,\ x' = x-1,\ y' = y, & & z' = z \\
\rho_{\tau_2}: & & y < z,\ x' = x+1,\ y' = y, & & z' = z-1 \\
\rho_{\tau_3}: & & y \geq z,\ x' = x, & y' = x+y, & z' = z \\
\rho_{\tau'_3}: & y \geq 1,\ & y \geq z,\ x' = x, & y' = x+y, & z' = z
\end{array}
$$

Fig. 2. Evolution of the termination transition system: initially (a) and after the first (b), second (c) and third (d) round.

Example 1: Let us show a termination analysis of the program in Fig. 1. In the first round, the solver finds the invariant $y \geq 1$ at ℓ_2 and the ranking function z for τ_2. While $y \geq 1$ can be added to τ_3 (resulting into a new transition τ'_3), the ranking function allows eliminating τ_2 from the termination transition system (see Fig. 2 (b)).

In the second round, the solver cannot find a ranking function. However, thanks to the Max-SMT formulation, it can produce the quasi-ranking function x, which is non-increasing and strict decreasing for τ_1, but not bounded. This quasi-ranking function can be used to split transition τ_1 into two new transitions $\tau_{1.1}$ and $\tau_{1.2}$ as follows:

$$
\begin{array}{lllll}
\rho_{\tau_{1.1}}: & \boldsymbol{x \geq 0}, & y \geq 1, & x' = x-1, & y' = y, \quad z' = z \\
\rho_{\tau_{1.2}}: & \boldsymbol{x < 0}, & y \geq 1, & x' = x-1, & y' = y, \quad z' = z
\end{array}
$$

Then $\tau_{1.1}$ is immediately removed, since x is a ranking function for it. The current termination transition system is given in Fig. 2 (c).

In the third and final round, the termination implication $x < 0$ is generated at ℓ_2, together with the ranking function y for transition τ'_3. Note that the termination implication is crucial to prove the strict decrease of y for τ'_3, and that the previously generated invariant $y \geq 1$ at ℓ_2 is needed to ensure boundedness. Now τ'_3 can be removed, which makes the graph acyclic (see Fig. 2 (d)). This concludes the termination proof.

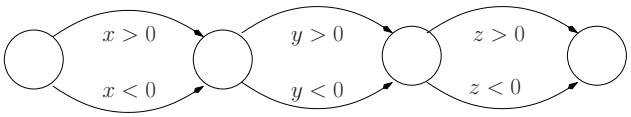


Fig. 3. Chain of locations obtained from a sequence of statements `assume`($x \neq 0$); `assume`($y \neq 0$); `assume`($z \neq 0$). Note disequalities are not natively supported, and so have to be split into disjunctions of inequalities.

IV. Implementation

The method presented in Sect. III has been implemented in the tool CppInv[1]. This section describes this implementation.

CppInv admits code written in C++ as well as in the language of T2 [10]. The system analyses programs with integer variables, linear expressions and function calls. Variables of other data types, such as floating-point variables, are treated as unknown values. Function calls are handled with techniques similar to those in [20], although currently the returned value is ignored. Further, for recursive functions, after a function call we assign unknowns to all variables that can be modified in the call (i.e., global variables and variables passed by reference).

In the transformation from the source code to the internal transition system representation, CppInv attempts to reduce the number of locations by composing transitions. Still, this preprocessing may result in an exponential growth in the number of transitions. As our technique does not require minimized transition systems for soundness, the tool stops this location minimization if a threshold number of transitions is reached. Moreover, whenever a chain of locations connected by transitions that do not modify variables (see Fig. 3) is detected, CppInv does not attempt to eliminate the locations: since no variable is updated, in these transitions any function satisfies the non-increase condition, while no ranking function is possible. For this reason, when producing the constraints, these transitions are ignored as far as termination is concerned, and are only considered for the generation of invariants.

Once the input is represented as a transition system, the actual termination analysis starts. See function *proved_TS_term*:

bool *proved_TS_term*(*Trans_Sys* $\mathcal{S} = (\overline{v},\ \mathcal{L},\ \Theta,\ \mathcal{T})$) {
  // $\mathcal{C}$ is the list of SCC's topologically sorted according to ordering $\prec$
  ($\mathcal{C}$, $\prec$) = *compute_SCCs_and_topologically_sort*($\mathcal{S}$);
  for ($C \in \mathcal{C}$ **by** $\prec$) {
    (L, T) = (*locations*(C), *transitions*(C));
    P = *copy*(T);
    for ($\ell \in L : \exists(\hat{\ell}, \ell, \rho) \in \mathcal{T}$ *with* $\hat{\ell} \in \widehat{C} \prec C$)
      $\Theta(\ell) = \Theta(\ell) \vee \mathrm{SPost}(\rho)$;
    if (**not** *proved_SCC_term*(L, T, P)) **return false**; }
  return true; }

The SCC's are computed and topologically sorted, and each SCC is processed according to this order. Processing an SCC involves first performing a copy of the transitions for keeping track of those not proven finitely executable yet. Then the initial conditions are updated with the strongest postconditions of the incoming transitions from previous SCC's, where the strongest postcondition of a transition relation $\rho(\overline{v}, \overline{v}')$ is the assertion $\mathrm{SPost}(\rho)(\overline{v}) \equiv \exists\, \overline{w}\ \ \rho(\overline{w}, \overline{v})$. Finally the SCC is analysed for termination. If it could not be proved terminating, the procedure stops. Otherwise the next SCC is dealt with.

The analysis of termination of SCC's is orchestrated by the function *proved_SCC_term*:

bool *proved_SCC_term*(*Set_Loc* L, *Set_Trans* T, *Set_Trans* P) {
  if (*dis_trans* (L, T, P) **or** *rank_fun*(L, T, P) **or** *term_impl*(L, T, P)){
    if (P == $\emptyset$) **return true**;
    for (C' *SCC in the graph of* P) {
      T' = *transitions*(C');
      if ($T' \neq \emptyset$ **and not** *proved_SCC_term*(L, T, T')) **return false**; }
    return true; }
  else return false ; }

It takes as arguments: a set of locations L and a set of transitions T, corresponding to an SCC of the transition system; and the *termination transition system*: a non-empty set $P \subseteq T$ of transitions that still have to be proved finitely executable. As explained in Sect. II-B, one may assume that the graph induced by P is strongly connected. The function returns **true** if all transitions in P can be proved finitely executable. We found out that, instead of directly solving the full constraint system introduced in Sect. III-B, in practice it is preferable to proceed by phases. Each phase[2] (functions *dis_trans*, *rank_fun* and *term_imp*) attempts to remove transitions from P by different means, and returns **true** if P has become empty or it is no longer strongly connected. In the former case, we are done. In the latter, the same procedure is recursively called. If after all phases P is non-empty, we report failure to prove termination.

In the first phase (function *dis_trans*), CppInv attempts to eliminate transitions with disability arguments by generating the appropriate invariants (neither ranking functions nor termination implications are considered at this point). This is achieved by solving the following Max-SMT formula: $\bigwedge_{\ell\in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau\in T}(\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge (\bigvee_{\tau\in T} \mathbb{D}_\tau \vee p_{\mathbb{D}}) \wedge [\neg p_{\mathbb{D}}, \omega_{\mathbb{D}}]$[3], where $p_{\mathbb{D}}$ is a propositional variable meaning that no transition can be disabled, and $\omega_{\mathbb{D}}$ is the corresponding weight. Transitions that are detected to be disabled (by means of a call to an SMT solver) are removed both from the original and the termination transition system. Invariants are used to strengthen the transition relations as described in Sect. II-B. The process is repeated while new transitions can be disabled.

bool *dis_trans* (*Set_Loc* L, *Set_Trans* T, *Set_Trans* P) {
  $cont$ = **true**;
  while ($cont$) {
    $cont$ = **false**;
    for ($\tau = (\ell, \ell', \rho) \in P$)
      if (ρ *is* UNSAT) // τ is disabled
        (T, P) = ($T - \{\tau\}$, $P - \{\tau\}$);
    if (P == $\emptyset$) **return true**;
    $H = \bigwedge_{\ell\in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau\in T} (\mathbb{D}_\tau \vee \mathbb{C}_\tau) \wedge \bigvee_{\tau\in T} (\mathbb{D}_\tau \vee p_{\mathbb{D}})$;
    $S = [\neg p_{\mathbb{D}}, \omega_{\mathbb{D}}]$;
    (I, c) = *solve*($H \wedge S$); // I invariant map, c cost of solution
    if (c == ∞) **break**; // No solution to hard clauses
    for ($\ell \in L, (\ell, \ell', \rho) \in \mathcal{T}$) // Strengthen relation with invariant
      $\rho = \rho \wedge I(\ell)$;
    if (c == 0) $cont$ = **true**; }
  return not *is_strongly_connected* (P); }

[1] CppInv, together with all benchmarks used in the experimental evaluation of Sect. V, is available at www.lsi.upc.edu/~albert/cppinv-term-bin.tar.gz.

[2] These phases have a time limit in our implementation although this is not made explicit in the pseudo-code shown below.

[3] Constraints that avoid redundancy are not included for simplicity.

In the second phase (function *rank_fun*), the system eliminates transitions by using ranking functions as arguments (termination implications are not considered at this point). If the computed function R satisfies the non-increase property, then each of the transitions τ in the termination transition system is examined and either removed if R is a ranking function for τ, or split otherwise, as described in Sect. III-B.

bool *rank_fun*(*Set_Loc* L, *Set_Trans* T, *Set_Trans* P){
  while (**true**) {
    $H = \bigwedge_{\ell\in L} \mathbb{I}_\ell \wedge \bigwedge_{\tau\in T} \mathbb{C}_\tau \wedge \bigvee_{\tau\in P}((\mathbb{B}_\tau \vee p_{\mathbb{B}}) \wedge (\mathbb{S}_\tau \vee p_{\mathbb{S}})) \wedge \bigwedge_{\tau\in P} (\mathbb{N}_\tau \vee p_{\mathbb{N}})$
    $S = [\neg p_{\mathbb{B}}, \omega_{\mathbb{B}}] \wedge [\neg p_{\mathbb{S}}, \omega_{\mathbb{S}}] \wedge [\neg p_{\mathbb{N}}, \omega_{\mathbb{N}}]$;
    (I, R, c) = *solve*$(H \wedge S)$;
    if $(c == \infty)$ **return false**; // No solution to hard clauses
    for $(\ell \in L, (\ell, \ell', \rho) \in \mathcal{T})$ // Strengthen relation with invariant
      $\rho = \rho \wedge I(\ell)$
    for $(\tau = (\ell, \ell', \rho) \in P)$
      if (ρ *is* UNSAT) // τ is disabled
        $(T, P) = (T - \{\tau\}, P - \{\tau\})$;
    if (*non_increase*(R))
      for $(\tau \in P)$
        if (*bounded*(τ, R) **and** *strict_decrease*(τ, R)) $P = P - \{\tau\}$;
        else *split* (τ, R, P); // Splits τ
    if $(P == \emptyset$ **or not** *is_strongly_connected*$(P))$ **return true**; } }

The third and final phase (function *term_impl*, not detailed here for lack of space) is very similar to the previous one, with the difference that termination implications are also included.

As regards the constraints, we restrain ourselves to invariants and ranking functions with *integer* coefficients, since this allows us to exploit efficient non-linear solving techniques [21]. Moreover, in order to perform integer reasoning, the following variation of Farkas' Lemma, based on the Gomory-Chvátal cutting plane rule [22], is employed:

Lemma 1: Let $Ax+b \leq 0$ ($A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$) be a system of linear inequalities over integer variables $x^T = (x_1, \ldots, x_n)$, and $c^T x+d \leq 0$ ($c \in \mathbb{Z}^n, d \in \mathbb{R}$) be a linear inequality. If there is $\lambda \in \mathbb{R}^m$, $i \in \mathbb{Z}$ and $f \in \mathbb{R}$ such that $\lambda \geq 0$, $c^T = \lambda^T A$, $\lambda^T b = i - f$, $0 \leq f < 1$ and $i \geq d$, then $Ax + b \leq 0$ entails $c^T x + d \leq 0$.

Lemma 1 allows transforming an $\exists\forall$ problem into an $\exists$ problem. If all coefficients in the premise are known values, the resulting satisfiability problem is an SMT problem over LA. Otherwise, an SMT problem over NA is obtained. Furthermore, as some unknowns are integer (the coefficients) and some are real (the multipliers), the resulting problems have mixed types.

CppInv uses Barcelogic [23] for solving the generated constraints. The Max-SMT(NA) solver for mixed non-linear arithmetic in Barcelogic extends the techniques presented in [21] for solving SMT(NIA) problems. This is achieved by allowing integer and real variables in the underlying linear arithmetic solver, and wrapping this solver with a branch-and-bound scheme for optimization [18].

V. Experimental Evaluation

In this section we show experiments that evaluate the performance of CppInv on a wide set of examples, which have been taken from the online programming learning environment Jutge.org [7] (see www.jutge.org), and from benchmark suites in [8] and in research.microsoft.com/en-us/projects/t2/. We provide here a comparison with the new version of T2, which according to the results given in [8] is performing much better when proving termination than most of the existing tools, including Terminator [12], AProVE [25] or ARMC [24], among others. We have also tried CProver [13] and Loopfrog [14], but the results were not good on these sets of benchmarks. All experiments were performed on an Intel Core i7 with 3.40GHz clock speed and 16 GB of RAM.

TABLE I
Results with benchmarks from T2

	#ins.	noMS	MS	MS+QR	MS+QR+TI	T2
Set1	449	212	220	228	238	245
Set2	472	245	252	262	276	279

TABLE II
Results with benchmarks from Jutge.org.

	#ins.	CppInv	T2
P11655	367	324	328
P12603	149	143	140
P12828	783	707	710
P16415	98	81	81
P24674	177	171	168
P33412	603	478	371

	#ins.	CppInv	T2
P40685	362	324	329
P45965	854	780	793
P70756	280	243	235
P81966	3642	2663	926
P82660	196	174	177
P84219	413	325	243

The first two considered sets of benchmarks are those provided by the T2 developers. Following the experiments in [8], we have set a 300 secs. timeout. To show the impact of the different techniques described in the paper, Table I presents the number of instances in each set (#ins.) and the number of those that we proved terminating with the following settings:

- *(noMS)* implements the generation of invariants and ranking functions using a translation to SMT(NA), but without using Max-SMT, i.e. with all constraints *hard*. The fact that this plain version can already prove many instances hints on the goodness of our underlying algorithm and the impact of using our NA-solver in this application.
- *(MS)* implements the generation of invariants and ranking functions using Max-SMT(NA), where the constraints imposed by the ranking function are added as *soft*.
- *(MS+QR)* adds to the previous case the possibility to use quasi-ranking functions.
- *(MS+QR+TI)* adds to the previous case the possibility to infer termination implications.

Note that every added improvement allows us to prove some more instances, while none is lost due to the additional complexity of the constraints generated.

Moreover, by looking into the results in more detail, we have observed that our tool and T2 complement each other to some extent: in Set1 CppInv can prove 7 instances which cannot be proved by T2, while we cannot prove 14 which can be handled by T2; similarly, in Set2 CppInv can prove 8 programs which cannot be proved by T2, while we cannot prove 11 that can be handled by T2. The average time in YES answers of T2 is 2.9 secs and of CppInv is 12.8 secs.

In Table II, we show the comparison of CppInv (with all described techniques) and T2 on our benchmarks from the programming learning environment Jutge.org, which is

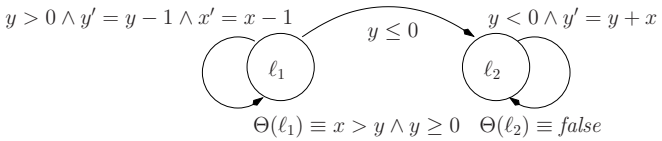


Fig. 4. Program that requires invariants from previous SCC's.

currently being used in several programming courses in the Universitat Politècnica de Catalunya. The benchmark suite consists of thousands of solutions written by students to 12 different programming problems. These programs can be considered challenging since most often they are not the most elegant solution but one with many more conditional statements than necessary (e.g., the largest instance we can successfully handle has nearly 700 transitions). Here, due to the size of the benchmark suites (see column #ins.), for the execution of both tools we have set a 120 secs. timeout. The average time in YES answers of T2 is 1.7 secs. and of CppInv is 1.6 secs. Note that, in order to run these benchmarks in T2, we have translated them into T2 format using our intermediate transition graph. This may be a disadvantage for T2, as it happens in the reverse way when CppInv is run on T2 benchmark set. In particular, we think the bad performance of T2 in sets P33412, P81966 and P84219 may be related to the way we handle division, which is crucial in these examples.

VI. Conclusions and Future Work

In short, the contributions of this paper are:

- a *novel Max-SMT constraint-based approach to proving termination*. Thanks to expressing the synthesis of a ranking function and a supporting invariant as a Max-SMT problem, we achieve a better guided and more fine-grained termination analysis than SMT-based methods. Max-SMT reveals to be a convenient framework for constraint-based termination analysis. In addition to our method, other techniques such as *unaffecting score maximization* [10] can be naturally modeled in Max-SMT.
- a *prototype of termination analyzer for (a subset of)* C++.

One of the shortcomings of our approach is that invariant synthesis is restricted to a single SCC. If invariants from previous SCC's have not been generated but are later required, our technique cannot prove termination. E.g., in the program shown in Fig. 4, the invariant $x > 0$ must be discovered at ℓ_1 so as to prove that the rightmost transition is finitely executable, although it is not necessary for proving that the leftmost loop is terminating. For future work we plan to develop techniques to overcome this kind of situations. A promising idea is to consider initiation conditions as soft: then the generated *quasi-invariants* represent what is missing from previous SCC's, and then can be propagated backwards. Alternatively, these quasi-invariants can be used to split the initial conditions of the current SCC. Finally, as a byproduct, this would allow us to solve the conditional termination problem as well.

Acknowledgment

This research was supported by Spanish MEC/MICINN under grant TIN 2010-21062-C02-01. We thank Jutge.org for providing benchmarks, and Byron Cook for giving us access to T2 and their benchmarks and for his helpful comments.

References

[1] D. Dams, R. Gerth, and O. Grumberg, "A heuristic for the automatic generation of ranking functions," in *Workshop on Advances in Verification*, 2000, pp. 1–8.
[2] M. Colón and H. Sipma, "Synthesis of linear ranking functions," in *TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 67–81.
[3] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *VMCAI*, ser. LNCS, vol. 2937. Springer, 2004, pp. 239–251.
[4] A. Tiwari, "Termination of linear programs," in *CAV*, ser. LNCS, vol. 3114. Springer, 2004, pp. 70–82.
[5] M. Colón and H. Sipma, "Practical methods for proving program termination," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 442–454.
[6] A. Bradley, Z. Manna, and H. Sipma, "Linear ranking with reachability," in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 491–504.
[7] J. Petit, O. Giménez, and S. Roura, "Jutge.org: an educational programming judge," in *SIGCSE*, ACM, 2012, pp. 445–450.
[8] M. Brockschmidt, B. Cook, and C. Fuhs, "Better termination proving through cooperation," in *CAV*, 2013, to appear.
[9] M. Colón, S. Sankaranarayanan, and H. Sipma, "Linear Invariant Generation Using Non-linear Constraint Solving," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 420–432.
[10] B. Cook, A. See, and F. Zuleger, "Ramsey vs. lexicographic termination proving," in *TACAS*, ser. LNCS, vol. 7795. Springer, 2013, pp. 47–61.
[11] A. Podelski and A. Rybalchenko, "Transition invariants," in *LICS*. IEEE Computer Society, 2004, pp. 32–41.
[12] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *PLDI*, ACM, 2006, pp. 415–426.
[13] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening, "Loop summarization and termination analysis," in *TACAS*, ser. LNCS, vol. 6605. Springer, 2011, pp. 81–95.
[14] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. Wintersteiger, "Loopfrog: A Static Analyzer for ANSI-C Programs," in *ASE*, IEEE, 2009, pp. 668–670.
[15] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv, "Proving conditional termination," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 328–340.
[16] P. Ganty and S. Genaim, "Proving termination starting from the end," in *CAV*, 2013, to appear.
[17] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.
[18] R. Nieuwenhuis and A. Oliveras, "On SAT Modulo Theories and Optimization Problems," in *SAT*, ser. LNCS, vol. 4121. Springer, 2006, pp. 156–169.
[19] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *ESOP*, ser. LNCS, vol. 3444. Springer, 2005, pp. 5–20.
[20] B. Cook, A. Podelski, and A. Rybalchenko, "Summarization for termination: no return!" *Formal Methods in System Design*, vol. 35, no. 3, pp. 369–387, 2009.
[21] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "SAT Modulo Linear Arithmetic for Solving Polynomial Constraints," *J. Autom. Reasoning*, vol. 48, no. 1, pp. 107–131, 2012.
[22] J. A. Robinson and A. Voronkov, Eds., *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.
[23] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The Barcelogic SMT Solver," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 294–298.
[24] A. Podelski and A. Rybalchenko, "ARMC: the logical choice for software model checking with abstraction refinement," in *PADL*, ser. LNCS, vol. 4354. Springer, 2007, pp. 245–259.
[25] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl, "Automated termination analysis of java bytecode by term rewriting," in *RTA* Volume 6 of LIPIcs., Schloss Dagstuhl, 2010, 259–276.