Science of Computer Programming 102 (2015) 142–157

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Symbolic execution as a basis for termination analysis [☆]

Germán Vidal

MiST, DSIC, Universitat Politècnica de València, E-46022, Valencia, Spain

ARTICLE INFO

Article history:
Received 24 April 2014
Received in revised form 1 November 2014
Accepted 29 January 2015
Available online 4 February 2015

Keywords:
Program termination
Symbolic execution
Program analysis

ABSTRACT

Program termination is a relevant property that has been extensively studied in the context of many different formalisms and programming languages. Traditional approaches to proving termination are usually based on inspecting the source code. Recently, a new semantics-based approach has emerged, which typically follows a two-stage scheme: first, a finite data structure representing the computation space of the program is built; then, termination is analyzed by inspecting the transitions in this data structure using traditional, syntax-based techniques.

Unfortunately, this approach is still specific to a programming language and semantics. In this work, we present instead a general, high-level framework that follows the semantics-based approach to proving termination. In particular, we focus on the first stage and advocate the use of symbolic execution, together with appropriate subsumption and abstraction operators, for producing a finite representation of the computations of a program. Hopefully, this higher level approach will provide useful insights for designing new semantics-based termination tools for particular programming languages.

1. Introduction

As witnessed by the extensive literature on the subject, determining whether a program terminates for all input data is a fundamental problem in computer science (see, e.g., [14,17,18,40], and references therein). In general, the techniques for proving program termination are specific to a programming language. Traditional approaches often rely on inspecting the *shape* of the program. For instance, one of the most popular approaches to analyzing termination of rewrite systems, the *dependency pairs* approach [7], is based on finding appropriate orderings that relate the left-hand side of the rules with some subterms of the right-hand side. *Size-change termination* analysis [28] for functional programs is also based on inspecting the source code, in this case finding an ordering that relates the size of the arguments of consecutive function calls. In the context of imperative programming, one can also find a flurry of works that are aimed at finding appropriate invariants to prove that all program loops are terminating by inspecting the source code (see e.g., [12,13,38,24] and references therein).

Despite the fact that some of these approaches are quite powerful, some drawbacks still exist. On the one hand, their syntax-driven nature makes them specific to a particular programming language and semantics. For example, the techniques developed for proving termination of *eager* functional programs and those for proving termination of *lazy* functional programs are different. Recently, a new semantics-based approach has emerged. Intuitively speaking, rather than inspecting the source code, this approach is based on constructing a finite representation of all the computations of a program—usually

☆ This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

E-mail address: gvidal@dsic.upv.es.

http://dx.doi.org/10.1016/j.scico.2015.01.007
0167-6423/© 2015 Elsevier B.V. All rights reserved.

a graph—and, then, inspecting the transitions in this graph in order to analyze the termination of the program. Actually, it suffices to restrict the analysis to those transitions that belong to a loop of the graph, i.e., a (potential) loop of the program. The construction of the graph can be seen as a front-end that depends on the considered programming language and semantics. Once the graph is built, one can express its transitions in a common language (e.g., a rule-based formalism such as term rewriting or logic programming) and, then, apply the existing syntax-directed approaches. In this way, the back-end could be shared by the different termination provers. We call this approach *semantics-based* since the front-end (the construction of the graph) is driven by the *semantics* of the program rather than by its syntax.

In the literature, we can already find a number of approaches that mostly follow this semantics-based scheme, e.g., to prove the termination of Haskell [21], Prolog with impure features [23,42], narrowing [35,46], or Java bytecode [2,3,9,10,36,43]. While all these approaches have proven useful in practice, they are still tailored to the specific features of the considered programming language and semantics. Unfortunately, this makes it rather difficult to grasp the key ingredients of the approach and, thus, it is not easy to design a termination tool for a different programming language by following the same pattern.

In this work, we present instead a general, high-level framework that follows the semantics-based approach to proving termination. Our purpose in this paper is not to introduce a new, semantics-based termination prover but to present a language-independent formulation that uses well-known principles from symbolic execution and partial evaluation, so that the vast literature on these subjects can be reused. In particular, we focus on the first stage—constructing a finite representation of all program computations—and advocate the use of symbolic execution, together with appropriate subsumption and abstraction operators. Symbolic execution [26,11] is a well-known technique for program verification, testing, debugging, etc. In contrast to concrete execution, symbolic execution considers that the values of some input data are unknown, i.e., some input parameters $x, y, \ldots$ take *symbolic values* $X, Y, \ldots$. Because of this, symbolic execution is often non-deterministic: at some control statements, we need to follow more than one execution path because the available information does not suffice to determine the validity of a control expression, e.g., symbolic execution may follow both branches of the conditional "`if` $(x>0)$ `then` $exp1$ `else` $exp2$" when the symbolic value X of variable x is not constrained enough to imply neither $x>0$ nor $\neg(x>0)$. Symbolic states include a *path condition* that stores the current constraints on symbolic values, i.e., the conditions that must hold to reach a particular execution state. For instance, after symbolically executing the above conditional, the derived states for $exp1$ and $exp2$ would add the conditions $X>0$ and $X\leq 0$, respectively, to their path conditions.

Traditionally, formal techniques based on symbolic execution have enforced soundness, i.e., the definition of *underapproximations*: if a symbolic state is reached and its path condition is satisfiable, there must be a concrete execution path that reaches the corresponding concrete state. These approaches, however, are often incomplete, i.e., there are some concrete computations that are not covered by the symbolic executions. In contrast, we consider a complete approach to symbolic execution, so that it *overapproximates* concrete execution. While underapproximations are useful for testing and debugging (since there are no false positives), overapproximations are important for verifying *liveness* properties such as program termination.

A preliminary version of some of the ideas in this paper appeared in [47]. Basically, [47] proposed a method for proving program termination by i) first producing a finite—but complete—symbolic execution of a program, ii) extracting a rewrite system that reproduces the transitions of symbolic execution and, finally, iii) using an off-the-shelf tool for proving the termination of the rewrite system (i.e., AProVE [22]). Here, in contrast to [47], we focus only on the front-end—constructing a symbolic execution graph—but introduce a more detailed approach. On the one hand, we consider a generalized notion of abstraction (w.r.t. [47]) which is much more useful in practice. Also, we introduce an algorithm for the construction of a finite representation of symbolic executions and define concrete subsumption and abstraction operators. Finally, we also prove the correctness of the resulting finite symbolic execution graphs (using the generic operations of subsumption and—generalized—abstraction), which guarantee their usefulness in the context of termination analysis.

The remainder of this paper is organized as follows. Section 2 introduces the notion of *complete* symbolic execution. We illustrate our developments using both a simple imperative language and a first-order eager functional language. Section 3 presents appropriate subsumption and abstraction operators so that a finite (but still complete) representation of the computation space can be obtained. Finally, Section 4 discusses some related work and Section 5 concludes.

2. Complete symbolic execution

In this section, we recall the notion of symbolic execution and introduce the conditions for completeness. We illustrate our developments with two simple programming languages.

Analogously to, e.g., [33], we abstract away from the syntax of a concrete programming language and represent a *program* P by a transition system denoted by a tuple $\langle \Sigma, \Theta, \mathcal{T}, \rho\rangle$ where Σ is a (possibly infinite) set of states, $\Theta \subseteq \Sigma$ are the initial states, $\mathcal{T}$ is a finite set of transitions (can be thought of as labels of program statements), and ρ is a mapping that assigns to each transition a binary relation over states: $\rho_\tau \subseteq \Sigma \times \Sigma$, for $\tau \in \mathcal{T}$. The *transition relation* of a program P is denoted by R_P and defined as follows:

$$R_P = \bigcup_{\tau \in \mathcal{T}} \rho_\tau$$

IMP

Expressions: include variables, integers, Booleans and the usual arithmetic and relational expressions.
Statements: assignments ($var := exp$), conditionals (**if** *cond* **then** *statement*1 [**else** *statement*2] **fi**) and iterations (**while** *cond* **do** *statements* **done**). Statements can be labeled. We also use **input**() to denote a generic I/O operation that reads data from the keyboard, file, etc.
States: $\langle l, \sigma\rangle$, where l is a program label and σ is a substitution for the program variables; here, pc denotes the program counter.
Transitions: transitions are associated to the program statements, so that the execution of a statement produces a state transition.
Transition relations: transition relations (s, s') can be compactly described as logical formulas over unprimed and primed variables corresponding to the variables of s and s' (so that variables not appearing in the formula are simply not constrained).

FUNC

Expressions: include variables, data constructors (e.g., list constructors Nil and Cons) and defined functions.
Statements: equations $lhs = rhs$, where lhs and rhs are expressions and $\mathcal{V}ar(rhs) \subseteq \mathcal{V}ar(lhs)$. We further require lhs to be of the form $f(t_1, \ldots, t_n)$, where f is a defined function and $t_1, \ldots, t_n$ are constructor terms (i.e., expressions built from variables and constructor symbols), and no variable occurs twice in lhs.
States: in this language, states are naturally associated to expressions.
Transitions: transitions are associated to the program equations, so that the application of an equation produces a state transition.
Transition relation: for every equation $lhs = rhs$, we have an associated transition relation of the form $(C[lhs\sigma], C[rhs\sigma])$ denoting the possible reductions using this rule. Here, $C[\,]$ denotes an arbitrary context and σ is an arbitrary substitution such that $\mathcal{V}ar(lhs\sigma) = \mathcal{V}ar(rhs\sigma) = \emptyset$.

Fig. 1. Example programming languages **IMP** and **FUNC**.

```
l0 :  x := input();
l1 :  while x > 0 do
l2 :      x := x − 1;
l3 :  done
```

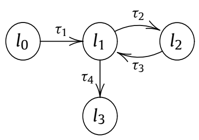


Fig. 2. Program WHILE and its control flow graph.

Thus, transition relations are (possibly infinite) sets of pairs of states (s, s'), where s is the current state and s' is the next state. We do not formalize the structure of states since it depends on the considered programming language (see Fig. 1).

In the following, we consider two simple programming languages for illustrating our developments: **IMP** (an imperative language) and **FUNC** (a call-by-value functional language). They are succinctly described in Fig. 1. In this work, we use *substitutions* to denote (finite) mappings from variables to values or expressions, together with the usual operations on substitutions: application of a substitution to an expression, composition of substitutions, etc. We denote the application of a substitution θ to an expression e by $e\theta$ (rather than $\theta(e)$). Given a state s and a substitution θ, by abuse of notation, we let $s\theta$ denote the application of θ to all expressions in s. The variables of a syntactic object o are denoted by $\mathcal{V}ar(o)$. Moreover, in the functional language, we use *contexts*, i.e., expressions containing a "hole", denoted by $C[\,]$, so that $C[e]$ is the expression that results from filling this hole with the expression e.

Computations are (possibly infinite) maximal sequences of states $s_0, s_1, \ldots$ such that

- $s_0 \in \Theta$ is an initial state and
- $(s_i, s_{i+1}) \in R_P$ for all $i \geq 0$ (up to the length of the sequence if it is finite).

We will denote computations as follows: $s_0 \xrightarrow{\tau_1}_{R_P} s_1 \xrightarrow{\tau_2}_{R_P} \ldots$ (we will omit the transition label and/or the program's transition relation when they are clear from the context, or are not relevant).

Example 1. Let us first consider the language **IMP** and the program WHILE shown in Fig. 2, with labels l_0, l_1, l_2, l_3 and only one variable x (besides the program counter pc). Here, four transitions are possible, τ_1 (associated to $x := \textbf{input}()$), τ_2 and τ_4 (associated to **while** $x > 0$ **do**...**done**) and τ_3 (associated to $x := x - 1$), as shown in the control flow graph depicted in Fig. 2. The transition relations can be defined as follows:

$$\begin{aligned}
\rho_{\tau_1} &: \; pc = l_0 \wedge pc' = l_1 \\
\rho_{\tau_2} &: \; pc = l_1 \wedge pc' = l_2 \wedge x > 0 \\
\rho_{\tau_3} &: \; pc = l_2 \wedge pc' = l_1 \wedge x' = x - 1 \\
\rho_{\tau_4} &: \; pc = l_1 \wedge pc' = l_3 \wedge x \leq 0
\end{aligned}$$

Therefore, the transition relation R_{WHILE} is defined as follows: $R_{\mathsf{WHILE}} = \rho_{\tau_1} \cup \rho_{\tau_2} \cup \rho_{\tau_3} \cup \rho_{\tau_4}$. For instance, given an initial state $\langle l_0, \{\,\}\rangle$, an example computation follows:

$$\begin{array}{lll}
r_0: \mathsf{main}(x,y) & = & \mathsf{mult}(x,y) \\
r_1: \mathsf{mult}(\mathsf{Z},y) & = & \mathsf{Z} \\
r_2: \mathsf{mult}(\mathsf{S}(x),y) & = & \mathsf{add}(\mathsf{mult}(x,y),y) \\
r_3: \mathsf{add}(\mathsf{Z},y) & = & y \\
r_4: \mathsf{add}(\mathsf{S}(x),y) & = & \mathsf{S}(\mathsf{add}(x,y))
\end{array}$$

Fig. 3. Program MULT.

$$\begin{array}{l}
\langle l_0, \{\,\}\rangle \overset{\tau_1}{\to} \langle l_1, \{x \mapsto 2\}\rangle \overset{\tau_2}{\to} \langle l_2, \{x \mapsto 2\}\rangle \\
\quad\quad\ \overset{\tau_3}{\to} \langle l_1, \{x \mapsto 1\}\rangle \overset{\tau_2}{\to} \langle l_2, \{x \mapsto 1\}\rangle \\
\quad\quad\ \overset{\tau_3}{\to} \langle l_1, \{x \mapsto 0\}\rangle \overset{\tau_4}{\to} \langle l_3, \{x \mapsto 0\}\rangle
\end{array}$$

Example 2. Let us now consider the language **FUNC** and the program MULT shown in Fig. 3. Here, mult (multiplication) and add (addition) denote defined functions while Z (zero) and S (successor) denote constructor symbols, used to build natural numbers. For simplicity, we consider that programs contain a special function, called main, that reduces to the initial expression to be evaluated.

Basically, standard functional reduction proceeds as follows: given an equation of the form $f(t_1,\ldots,t_n) = r$ and an expression e, we should find an innermost function call of the form $f(s_1,\ldots,s_n)$, i.e., $e = C[f(s_1,\ldots,s_n)]$, such that there exists a matching substitution σ with $f(t_1,\ldots,t_n)\sigma = f(s_1,\ldots,s_n)$; then, we have that $C[f(s_1,\ldots,s_n)]$ reduces to $C[r\sigma]$. Consequently, we have a transition associated to each (labeled) equation, whose transition relations are defined as follows:

$$\begin{array}{l}
\rho_{r_0}: (C[\mathsf{main}(x,y)\sigma], C[\mathsf{mult}(x,y)\sigma]) \\
\rho_{r_1}: (C[\mathsf{mult}(\mathsf{Z},y)\sigma], C[\mathsf{Z}\sigma]) \\
\rho_{r_2}: (C[\mathsf{mult}(\mathsf{S}(x),y)\sigma], C[\mathsf{add}(\mathsf{mult}(x,y),y)\sigma]) \\
\rho_{r_3}: (C[\mathsf{add}(\mathsf{Z},y)\sigma], C[y\sigma]) \\
\rho_{r_4}: (C[\mathsf{add}(\mathsf{S}(x),y)\sigma], C[\mathsf{S}(\mathsf{add}(x,y))\sigma])
\end{array}$$

for any constructor substitution σ (i.e., a substitution from variables to terms made of constructor symbols) that makes the expressions ground (i.e., without variables). Therefore, the transition relation R_{MULT} is now defined by $R_{\mathsf{MULT}} = \rho_{r_0} \cup \rho_{r_1} \cup \rho_{r_2} \cup \rho_{r_3} \cup \rho_{r_4}$. For instance, given the initial expression $\mathsf{main}(\mathsf{S}(\mathsf{Z}),\mathsf{S}(\mathsf{S}(\mathsf{Z})))$, we have the following computation:

$$\begin{array}{rl}
\mathsf{main}(\mathsf{S}(\mathsf{Z}),\mathsf{S}(\mathsf{S}(\mathsf{Z}))) & \overset{r_0}{\to} \mathsf{mult}(\mathsf{S}(\mathsf{Z}),\mathsf{S}(\mathsf{S}(\mathsf{Z}))) \\
& \overset{r_2}{\to} \mathsf{add}(\mathsf{mult}(\mathsf{Z},\mathsf{S}(\mathsf{S}(\mathsf{Z}))),\mathsf{S}(\mathsf{S}(\mathsf{Z}))) \\
& \overset{r_1}{\to} \mathsf{add}(\mathsf{Z},\mathsf{S}(\mathsf{S}(\mathsf{Z}))) \\
& \overset{r_3}{\to} \mathsf{S}(\mathsf{S}(\mathsf{Z}))
\end{array}$$

Therefore, we have that $\mathsf{main}(\mathsf{S}(\mathsf{Z}),\mathsf{S}(\mathsf{S}(\mathsf{Z})))$ reduces to $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$.

Symbolic execution [26,11], originally introduced in the context of program testing and debugging, extends concrete execution in order to deal with variables bound to symbolic expressions (instead of concrete values). For instance, $\langle l_0, \{x \mapsto X, y \mapsto Y, z \mapsto 42\}, X > Y\rangle$ is a symbolic state in the language **IMP**, where x, y are program variables bound to symbolic values (denoted by capital letters), z is a local variable bound to the integer 42, and $X > Y$ is a *path condition* (see Fig. 1). Program variables can also be bound to symbolic *expressions* like $X + 2 * Y$ or arbitrary data structures (e.g., arrays, linked lists, etc.) possibly including symbolic values denoting missing information. Control statements often involve (non-deterministically) exploring several paths. The *path condition* of symbolic states is then used to keep track of the assumptions made on the symbolic values in each computation thread.

In the following, we denote symbolic states with $\mathcal{S}_1$, $\mathcal{S}_2$, etc.; also, we use $\Sigma^\sharp$ to denote the domain of symbolic states. Here, we assume that the structure of symbolic states is unknown but shares the same elements of concrete states (replacing values with symbolic expressions if needed) plus a path condition. Let us now introduce some useful auxiliary functions:

Definition 1. Given a symbolic state $\mathcal{S}$, we introduce the following functions:

- $\mathsf{pcond}(\mathcal{S})$ denotes the path condition of $\mathcal{S}$;
- $\mathsf{svars}(\mathcal{S})$ denotes the set of symbolic variables that occur in $\mathcal{S}$;
- $\mathsf{state}(\mathcal{S})$ denotes a state that is equal to $\mathcal{S}$ but it does not include the path condition (but may contain some symbolic expressions and not only values).

Moreover, let V be a set of symbolic variables and π be a path condition:

- $\mathsf{sols}_V(\pi)$ denotes the set of substitutions for the variables of V that satisfy π.

There is a clear relation between concrete and symbolic states: a symbolic state represents the set of concrete states that can be obtained by replacing its symbolic variables with concrete values such that the path condition holds:

Definition 2 *(Concretization).* Given a symbolic state, $\mathcal{S}$, with $\pi = \mathsf{pcond}(\mathcal{S})$, $V = \mathsf{svars}(\mathcal{S})$ and $s = \mathsf{state}(\mathcal{S})$, the concretization function $\gamma : \Sigma^\sharp \mapsto \wp(\Sigma)$, returns $\gamma(\mathcal{S}) = \{\overline{s\theta} \mid \theta \in \mathsf{sols}_V(\pi)\}$, where $\overline{s\theta}$ denotes the state that results from $s\theta$ by evaluating its expressions (if any).

Let us note that our symbolic states share some similarities with the notion of *region* in [25].

Example 3. In the language **IMP**, symbolic states can be denoted by $\langle l, \theta, \pi\rangle$, where l is a label, θ is a substitution from program variables to *symbolic expressions*, and π is a path condition. Given, for instance, a symbolic state $\mathcal{S} = \langle l_2, \{x \mapsto X-1\}, X > 1\rangle$, we have $\mathsf{pcond}(\mathcal{S}) = X > 1$, $\mathsf{svars}(\mathcal{S}) = \{X\}$, $\mathsf{state}(\mathcal{S}) = \langle l_2, \{x \mapsto X-1\}\rangle$ and $\mathsf{sols}_{\{X\}}(X > 1) = \{\{X \mapsto 2\}, \{X \mapsto 3\}, \ldots\}$. Therefore, we have

$$\gamma(\mathcal{S}) = \{\overline{\langle l_2, \{x \mapsto 2-1\}\rangle}, \overline{\langle l_2, \{x \mapsto 3-1\}\rangle}, \ldots\} = \{\langle l_2, \{x \mapsto 1\}\rangle, \langle l_2, \{x \mapsto 2\}\rangle, \ldots\}$$

In the language **FUNC**, symbolic states can be denoted by $\langle e, \theta\rangle$, where e is an expression, possibly containing symbolic variables, and θ is a substitution from symbolic variables to symbolic expressions. Given for instance a symbolic state

$$\mathcal{S} = \langle \mathsf{add}(\mathsf{mult}(W, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \{X \mapsto \mathsf{S}(W)\}\rangle$$

we have $\mathsf{pcond}(\mathcal{S}) = \{X \mapsto \mathsf{S}(W)\}$ (i.e., the path condition is now represented by a substitution), $\mathsf{svars}(\mathcal{S}) = \{X, W\}$, $\mathsf{state}(\mathcal{S}) = \mathsf{add}(\mathsf{mult}(W, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \mathsf{S}(\mathsf{S}(\mathsf{Z})))$ and

$$\mathsf{sols}_{\{X,W\}}(\{X \mapsto \mathsf{S}(W)\}) = \{\{X \mapsto \mathsf{S}(\mathsf{Z}), W \mapsto \mathsf{Z}\}, \{X \mapsto \mathsf{S}(\mathsf{S}(\mathsf{Z})), W \mapsto \mathsf{S}(\mathsf{Z})\}, \ldots\}$$

Hence, $\gamma(\mathcal{S}) = \{\mathsf{add}(\mathsf{mult}(\mathsf{Z}, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \mathsf{S}(\mathsf{S}(\mathsf{Z}))),\ \mathsf{add}(\mathsf{mult}(\mathsf{S}(\mathsf{Z}), \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \ldots\}$.[1]

In the following, we assume a *decidable* partial order $\sqsubseteq_\gamma$ on symbolic states:

Definition 3. We denote by $\sqsubseteq_\gamma$ a partial order on symbolic states such that $\mathcal{S} \sqsubseteq_\gamma \mathcal{S}'$ implies $\gamma(\mathcal{S}) \subseteq \gamma(\mathcal{S}')$. Moreover, we assume that $(\Sigma^\sharp, \sqsubseteq)$ is a partial ordered set (poset) with the *ascending chain condition*: every ascending sequence of the form $\mathcal{S}_0 \sqsubseteq \mathcal{S}_1 \sqsubseteq \mathcal{S}_2 \sqsubseteq \ldots$ eventually stabilizes, i.e., there exists a positive integer n such that $\mathcal{S}_n = \mathcal{S}_{n+1} = \mathcal{S}_{n+2} = \ldots$.

Essentially, the ascending chain condition is required for the *abstraction operator* to be safe. Loosely speaking, given a poset $(\Sigma^\sharp, \sqsubseteq)$ and an abstraction operator α (see Section 3.2 for a precise definition), we require $\mathcal{S} \sqsubseteq \alpha(\mathcal{S})$ for each symbolic state $\mathcal{S}$, i.e., we require $\alpha(\mathcal{S})$ to describe more concrete states than $\mathcal{S}$. When $(\Sigma^\sharp, \sqsubseteq)$ satisfies the ascending chain condition, the application of abstraction allows us to always get a symbolic state $\mathcal{S}$ such that $\alpha(\mathcal{S}) = \mathcal{S}$ so that it cannot be further generalized (which is essential to guarantee the completeness of the process).

The ascending chain condition is usually required in the context of *abstract interpretation* [15]. When the poset is finite, it trivially satisfies the ascending chain condition. When it is infinite, we can combine abstraction with a *widening operator* so that the resulting partial order satisfies the ascending chain condition (see, e.g., [16]). For instance, given a domain of intervals of integer numbers and a finite set T (called the threshold set), one can define a widening operator $\triangledown_T$ as follows:

$$\begin{array}{rcl} [a,b] \triangledown_T [a',b'] & = & [\text{if } a' < a \text{ then } max\{l \in T \mid l \leq a\} \\ & & \qquad\qquad\ \text{else } a \\ & & \ \ \text{if } b' > b \text{ then } min\{h \in T \mid h \geq b'\} \\ & & \qquad\qquad\ \text{else } b] \end{array}$$

For example, one can consider the set $T = \{-\infty, 0, \infty\}$ so that $[0,10] \triangledown_T [1,10] = [1,10]$ but $[0,10] \triangledown_T [0,12] = [0,+\infty]$. Then, if the partial order induced by the abstraction operator α does not satisfy the ascending chain condition, one can use a widening operator like $\triangledown_T$ so that $\alpha'(\mathcal{S}) = \mathcal{S} \triangledown_T \alpha(\mathcal{S})$ and α' satisfies the ascending chain condition by construction [16].

For simplicity, in this paper we will only consider a finite domain of integers in the examples (from $-minint$ to $+maxint$). Nevertheless, the examples could be extended to deal with an infinite domain of integers by, e.g., introducing integer intervals and a widening operator as shown above.

Now, we introduce our notion of *symbolic program*, which is again defined as a transition system. Analogously to $\Sigma^\sharp$, we use $\Theta^\sharp$, $\mathcal{T}^\sharp$ and $\rho^\sharp$ to denote the initial symbolic states, the finite set of transitions, and the mapping that assigns to each transition a binary relation over symbolic states, respectively.

[1] The auxiliary function $\bar{e}$ to evaluate the symbolic expressions of e is not needed in this language since $\bar{e} = e$ for all expressions as long as built-in values and operators (e.g., integers and arithmetic operators) are not allowed.

Definition 4 *(Symbolic program).* Let $P = \langle \Sigma, \Theta, \mathcal{T}, \rho \rangle$ be a concrete program. We say that $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}^\sharp, \rho^\sharp \rangle$ is a symbolic version of P if the following conditions hold:

1. $\forall s \in \Sigma.\ \exists \mathcal{S} \in \Sigma^\sharp$ such that $s \in \gamma(\mathcal{S})$;
2. $\forall s \in \Theta.\ \exists \mathcal{S} \in \Theta^\sharp$ such that $s \in \gamma(\mathcal{S})$;
3. $\mathcal{T} = \mathcal{T}^\sharp$ (i.e., the program sentences are not changed);
4. $\forall (s, s') \in \rho_\tau$ and $\forall \mathcal{S} \in \Sigma^\sharp$ such that $s \in \gamma(\mathcal{S})$ there exists $(\mathcal{S}, \mathcal{S}') \in \rho^\sharp_\tau$ with $s' \in \gamma(\mathcal{S}')$.

Loosely speaking, conditions (1) and (2) imply that replacing some values by symbolic expressions do not change the nature of a state. Condition (3) means that symbolic execution does not change the source program (only the input data can be replaced by symbolic values). Finally, condition (4) states that symbolic execution is indeed an overapproximation of normal execution, which guarantees that all concrete transitions have a counterpart in the symbolic program (and this is essential to analyze program termination).

As before, the transition relation $R_{P^\sharp}$ of a symbolic program $P^\sharp$ is defined as the union of all transition relations: $R_{P^\sharp} = \bigcup_{\tau \in \mathcal{T}^\sharp} \rho^\sharp_\tau$. Symbolic executions are (possibly infinite) maximal sequences of symbolic states $\mathcal{S}_0, \mathcal{S}_1, \ldots$ such that

- $\mathcal{S}_0 \in \Theta^\sharp$ is an initial symbolic state and
- $(\mathcal{S}_i, \mathcal{S}_{i+1}) \in R_{P^\sharp}$ for all $i \geq 0$ (up to the length of the sequence if it is finite).

We will denote symbolic executions as follows:

$$\mathcal{S}_0 \overset{\tau_1,\pi_1}{\leadsto}_{R_{P^\sharp}} \mathcal{S}_1 \overset{\tau_2,\pi_2}{\leadsto}_{R_{P^\sharp}} \mathcal{S}_2 \overset{\tau_3,\pi_3}{\leadsto}_{R_{P^\sharp}} \ldots$$

where τ_i is the transition of the step and π_i are the *new* constraints that are added to the path condition (we will omit the transition label, the path condition, and/or the program's transition relation when they are clear from the context, or are not relevant). Here we consider that the satisfiability of the path condition is checked at every step. If the domain of path conditions is not decidable (as is often the case), one can use a time bound so that if the constraints are not solved within this bound, the path condition is assumed satisfiable (to ensure that an overapproximation is computed, which contrasts with traditional approaches where it is assumed *unsatisfiable* to preserve the generation of underapproximations).

Example 4. Consider again the program WHILE shown in Fig. 2. Let WHILE$^\sharp$ be a symbolic version of this program, where the symbolic transition relations are defined in the obvious way: when a conditional is reached, we follow all paths which are consistent with the current path condition, and update them with the new constraints. More precisely, let us consider the following simple statement:

$$\begin{array}{ll} l_1: & \textbf{while}\ x > 0\ \textbf{do} \\ l_2: & \quad x := x - 1; \\ l_3: & \textbf{done} \end{array}$$

Then, given a symbolic state $\langle l_1, \{x \mapsto X\}, X < 0 \rangle$, there is only one possible transition to $\langle l_3, \{x \mapsto X\}, X < 0 \rangle$ since the current path condition, $X < 0$, makes the condition $x > 0$ false. On the other hand, given the symbolic state $\langle l_1, \{x \mapsto X\}, true \rangle$, we would have two possible transitions: to the symbolic state $\langle l_2, \{x \mapsto X\}, X > 0 \rangle$ and to the symbolic state $\langle l_3, \{x \mapsto X\}, X \leq 0 \rangle$.

As an example computation, let us consider the initial symbolic state $\langle l_0, \{x \mapsto X\}, true \rangle$. Here, we have for instance the following symbolic execution:

$$\begin{array}{rl} \langle l_0, \{x \mapsto X\}, true \rangle \overset{\tau_1}{\leadsto} & \langle l_1, \{x \mapsto X\}, true \rangle \\ \overset{\tau_2}{\leadsto} & \langle l_2, \{x \mapsto X\}, X > 0 \rangle \\ \overset{\tau_3}{\leadsto} & \langle l_1, \{x \mapsto X - 1\}, X > 0 \rangle \\ \overset{\tau_2}{\leadsto} & \langle l_2, \{x \mapsto X - 1\}, X > 1 \rangle \\ \overset{\tau_3}{\leadsto} & \langle l_1, \{x \mapsto X - 2\}, X > 1 \rangle \\ \overset{\tau_4}{\leadsto} & \langle l_3, \{x \mapsto X - 2\}, X = 2 \rangle \end{array}$$

Note that we have simplified the path condition $X > 0 \wedge X - 1 > 0$ to $X > 1$ and the path condition $X > 1 \wedge X - 2 = 0$ to $X = 2$.

Example 5. Consider now the program MULT shown in Fig. 3. Let MULT$^\sharp$ be a symbolic version of this program; here, symbolic transitions are defined using a symbolic extension of functional reduction. Essentially, the symbolic extension will consider a *unifying* substitution instead of a matching substitution to allow the instantiation of symbolic variables in the expression being reduced: given an equation $f(t_1, \ldots, t_n) = r$ and an expression e, we should find an innermost function call of the form $f(s_1,\dots,s_n)$, i.e., $e = C[f(s_1,\dots,s_n)]$, such that there exists a unifying substitution σ with $f(t_1,\dots,t_n)\sigma = f(s_1,\dots,s_n)\sigma$; therefore, we now have that $C[f(s_1,\dots,s_n)]$ reduces to $(C[r])\sigma$.[2] More details can be found, e.g., in [4], where a symbolic extension of functional reduction is introduced for defining a partial evaluation framework.

For instance, given the initial symbolic state $\langle \mathsf{main}(X, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), id\rangle$, where X denotes a symbolic variable, we have the following symbolic execution:

$$\begin{array}{rl} \langle \mathsf{main}(X,\mathsf{S}(\mathsf{S}(\mathsf{Z}))), id\rangle & \overset{r_0}{\rightsquigarrow} \langle \mathsf{mult}(X,\mathsf{S}(\mathsf{S}(\mathsf{Z}))), id\rangle \\ & \overset{r_2}{\rightsquigarrow} \langle \mathsf{add}(\mathsf{mult}(W,\mathsf{S}(\mathsf{S}(\mathsf{Z}))),\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \{X \mapsto \mathsf{S}(W)\}\rangle \\ & \overset{r_1}{\rightsquigarrow} \langle \mathsf{add}(\mathsf{Z},\mathsf{S}(\mathsf{S}(\mathsf{Z}))), \{X \mapsto \mathsf{S}(\mathsf{Z}), W \mapsto \mathsf{Z}\}\rangle \\ & \overset{r_3}{\rightsquigarrow} \langle \mathsf{S}(\mathsf{S}(\mathsf{Z})), \{X \mapsto \mathsf{S}(\mathsf{Z}), W \mapsto \mathsf{Z}\}\rangle \end{array}$$

So we have that $\mathsf{main}(X,\mathsf{S}(\mathsf{S}(\mathsf{Z})))$ reduces to $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$ if X takes the value $\mathsf{S}(\mathsf{Z})$.

Now, we lift the fourth condition of Definition 4 to computations.

Lemma 1 *(Overapproximation). Let P be a program and $P^\sharp$ a symbolic version of P. If there exists a (possibly infinite) computation of the form $s_0 \xrightarrow{\tau_1}_{R_P} s_1 \xrightarrow{\tau_2}_{R_P} \dots$ then, for any symbolic state $\mathcal{S}_0$ such that $s_0 \in \gamma(\mathcal{S}_0)$, we have $\mathcal{S}_0 \overset{\tau_1}{\rightsquigarrow}_{R_P^\sharp} \mathcal{S}_1 \overset{\tau_2}{\rightsquigarrow}_{R_P^\sharp} \dots$ where $s_i \in \gamma(\mathcal{S}_i)$ for all $i > 0$.*

Proof. The claim follows straightforwardly by applying property (4) of Definition 4. Consider the first transition $s_0 \xrightarrow{\tau_1}_{R_P} s_1$. By property (4), we have that, for all $\mathcal{S}_0 \in \Theta^\sharp$ such that $s_0 \in \gamma(\mathcal{S}_0)$, the transition $\mathcal{S}_0 \overset{\tau_1}{\rightsquigarrow}_{R_P^\sharp} \mathcal{S}_1$ holds with $s_1 \in \gamma(\mathcal{S}_1)$. The same reasoning can be applied repeatedly so that a symbolic execution mimicking the transitions of the concrete computation is built. □

Therefore, we can conclude that any symbolic version of a program that fulfills the conditions of Definition 4 is complete, i.e., it produces an overapproximation of the concrete computations. Thus the termination of the original program can be analyzed by inspecting the symbolic executions.

3. Constructing a finite representation of symbolic executions

The main drawback of symbolic execution is that the computation space is usually infinite, even for programs where concrete executions always terminate. When computing underapproximations, this is not a significant drawback since one can just stop the symbolic executions at any point (e.g., introducing a bound on the number of symbolic execution steps). Unfortunately, this is not safe when computing an overapproximation. In this section, we present a framework for producing a finite representation of the (possibly infinite) symbolic executions of a program that is still complete.

Example 6. Consider again a symbolic version $\mathsf{WHILE}^\sharp$ of the program of Fig. 2. Here, we have for instance the following infinite symbolic execution:

$$\begin{array}{rl} \langle l_0, \{x \mapsto X\}, true\rangle & \overset{\tau_1}{\rightsquigarrow} \langle l_1, \{x \mapsto X\}, true\rangle \\ & \overset{\tau_2}{\rightsquigarrow} \langle l_2, \{x \mapsto X\}, X > 0\rangle \\ & \overset{\tau_3}{\rightsquigarrow} \langle l_1, \{x \mapsto X-1\}, X > 0\rangle \\ & \overset{\tau_2}{\rightsquigarrow} \langle l_2, \{x \mapsto X-1\}, X > 1\rangle \\ & \overset{\tau_3}{\rightsquigarrow} \langle l_1, \{x \mapsto X-2\}, X > 1\rangle \\ & \overset{\tau_2}{\rightsquigarrow} \dots \end{array}$$

by always choosing transition τ_2 from location l_1.

Analogously, we have the following infinite symbolic execution using the symbolic program $\mathsf{MULT}^\sharp$:

$$\begin{array}{rl} \langle \mathsf{add}(X,Y), id\rangle & \overset{r_4}{\rightsquigarrow} \langle \mathsf{S}(\mathsf{add}(X',Y)), \{X \mapsto \mathsf{S}(X')\}\rangle \\ & \overset{r_4}{\rightsquigarrow} \langle \mathsf{S}(\mathsf{S}(\mathsf{add}(X',Y))), \{X \mapsto \mathsf{S}(\mathsf{S}(X'')), X' \mapsto \mathsf{S}(X'')\}\rangle \\ & \overset{r_4}{\rightsquigarrow} \dots \end{array}$$

by always performing a transition using rule r_4.

[2] Observe that the context may contain symbolic variables that are instantiated by σ, thus we apply it to the whole expression and not only to r.

The computations of a symbolic program can be represented by means of a tree-like structure as follows:

Definition 5 *(Symbolic execution graph).* Let $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}, \rho^\sharp \rangle$ be a symbolic program. We represent the computations of $P^\sharp$ for an initial symbolic state $\mathcal{S}_0 \in \Theta^\sharp$ by means of a (possibly infinite) directed rooted node- and edge-labeled graph $\mathcal{G}_{P^\sharp}$:

- nodes are labeled with symbolic states from $\Sigma^\sharp$, with $\mathcal{S}_0$ the root node;
- edges are labeled with a pair τ, π (a transition from $\mathcal{T}$ and a logical formula denoting the new path condition added in the step);
- there is an edge labeled with τ, π from a node labeled with $\mathcal{S}$ to a node labeled with $\mathcal{S}'$, denoted by $\mathcal{S} \xrightarrow{\tau,\pi} \mathcal{S}'$, iff $\mathcal{S} \overset{\tau,\pi}{\leadsto}_{R_{P^\sharp}} \mathcal{S}'$ (we will ignore τ and/or π when they are clear from the context, or are not relevant);
- nodes can be marked (graphically denoted by underlining the node) or unmarked; initially, the root node is unmarked and nodes are marked when their symbolic execution is already considered. As we will see later, marks are used in the algorithm for constructing symbolic execution trees (cf. Definition 12) to keep track of the nodes that have been already processed.

Clearly, every symbolic program and initial symbolic state induce an associated symbolic execution graph which is generally infinite.

In the literature of symbolic execution, one can find two basic operations in order to make the symbolic execution graph finite: *subsumption* and *abstraction* (see, e.g., [5,6]). Similar operations can also be found in the partial evaluation literature (see, e.g., [32,34,30]).

3.1. Subsumption

In general, subsumption allows us to stop symbolic execution when we reach a state that is an *instance* of (i.e., it is *subsumed* by) a previous one.

In the following, we assume that symbolic execution graphs may also contain edges labeled with sub and abs to account for subsumption and abstraction steps.

Definition 6 *(Subsumption step).* Let $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}, \rho^\sharp \rangle$ be a symbolic program and $\mathcal{G}_{P^\sharp}$ be a (possibly incomplete) symbolic execution graph for $\mathcal{S}_0 \in \Theta^\sharp$ where

$$\underline{\mathcal{S}_0} \longrightarrow \underline{\mathcal{S}_1} \longrightarrow \ldots \longrightarrow \underline{\mathcal{S}_{n-1}} \longrightarrow \mathcal{S}_n$$

is a path in the graph with $n > 0$, so that all nodes are marked except the last one. If there exists a node labeled with $\mathcal{S}_i$, $0 \leq i < n$, such that $\mathcal{S}_i \longrightarrow \mathcal{S}_{i+1}$ is a symbolic execution step and $\mathcal{S}_n \sqsubseteq_\gamma \mathcal{S}_i$, we mark $\mathcal{S}_n$ and add an edge labeled with sub from $\mathcal{S}_n$ to $\mathcal{S}_i$.

Observe that our definition of subsumption is nondeterministic since there are two degrees of freedom: when subsumption is to be considered (e.g., at every node), and the choice of the previous nodes with which subsumption is tested (e.g., all ancestors in the same derivation). Here, we consider a simple strategy that is based on the notion of *comparable* states.

Definition 7 *(Comparison relation).* Let $\Sigma^\sharp$ be a domain of symbolic states and let $\sim$ be an equivalence relation for $\Sigma^\sharp$. As usual, we let $[s] = \{s' \in \Sigma^\sharp \mid s \sim s'\}$ denote the equivalence class of state s and $\Sigma^\sharp/\sim$ denote the set of equivalence classes of $\Sigma^\sharp$.

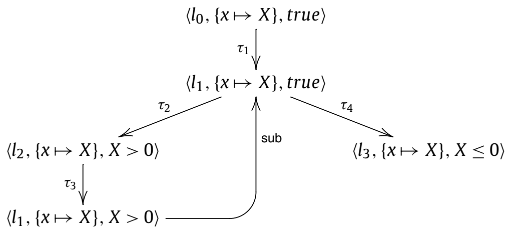
We say that $\sim$ is a *comparison relation* for $\Sigma^\sharp$ if it induces a finite number of equivalence classes, i.e., $\Sigma^\sharp/\sim$ is finite.

Intuitively speaking, restricting our tests to comparable states is safe by Ramsey's theorem [41] since the number of incomparable equivalence classes is finite.

In the following, we consider the following simple strategy: every new node in the symbolic execution graph is tested against all *comparable* ancestors: if the new node is subsumed by a previous one, a subsumption edge is added and the node is marked; otherwise, nothing is done and symbolic execution proceeds.

Example 7. Consider the symbolic program WHILE$^\sharp$ and the infinite computation shown in Example 6. Here, we consider that two symbolic states are comparable if they have the same label, i.e., $\langle l_1, \sigma_1, \pi_1 \rangle \sim \langle l_2, \sigma_2, \pi_2 \rangle$ iff $l_1 = l_2$. Then, the infinite-state path in the graph is made finite by using our subsumption strategy as follows[3]:

[3] Note that the fact that the symbolic execution is now finite does not imply the termination of concrete executions. Actually, the original program is not terminating unless the values of x are bounded by some minimum value.



$$\begin{tikzcd}
 & \langle l_0, \{x \mapsto X\}, true\rangle \arrow[d, "\tau_1"'] & \\
 & \langle l_1, \{x \mapsto X\}, true\rangle \arrow[dl, "\tau_2"'] \arrow[dr, "\tau_4"] & \\
\langle l_2, \{x \mapsto X\}, X > 0\rangle \arrow[d, "\tau_3"'] & & \langle l_3, \{x \mapsto X\}, X \leq 0\rangle \\
\langle l_1, \{x \mapsto X\}, X > 0\rangle \arrow[uur, bend right, "\mathsf{sub}"'] & &
\end{tikzcd}$$

Fig. 4. Symbolic execution graph for program $\mathsf{WHILE}^\sharp$.

$$\begin{aligned}
\langle l_0, \{x \mapsto X\}, true\rangle &\xrightarrow{\tau_1} \langle l_1, \{x \mapsto X\}, true\rangle \qquad\quad \equiv \mathcal{S}_2\\
&\xrightarrow{\tau_2} \langle l_2, \{x \mapsto X\}, X > 0\rangle\\
&\xrightarrow{\tau_3} \langle l_1, \{x \mapsto X-1\}, X > 0\rangle \equiv \mathcal{S}_4\\
&\xrightarrow{\mathsf{sub}} \mathcal{S}_2
\end{aligned}$$

since $\gamma(\mathcal{S}_4) = \{\langle l_1, \{x \mapsto 0\}\rangle, \langle l_1, \{x \mapsto 1\}\rangle, \ldots\} \subseteq \{\ldots, \langle l_1, \{x \mapsto -1\}\rangle, \langle l_1, \{x \mapsto 0\}\rangle, \langle l_1, \{x \mapsto 1\}\rangle, \ldots\} = \gamma(\mathcal{S}_2)$ and, thus, $\mathcal{S}_4 \sqsubseteq_\gamma \mathcal{S}_2$ would hold in any sensible definition of $\sqsubseteq_\gamma$. The symbolic execution graph for program $\mathsf{WHILE}^\sharp$ is shown in Fig. 4.

Example 8. Let us consider now the symbolic program $\mathsf{MULT}^\sharp$. Here, we consider that two states $\langle e_1, \sigma_1\rangle$ and $\langle e_2, \sigma_2\rangle$ are comparable if the expressions e_1 and e_2 are rooted by the same (defined or constructor) function symbol. In this language, $\langle e_2, \sigma_2\rangle \sqsubseteq_\gamma \langle e_1, \sigma_1\rangle$ if $e_2\sigma_2$ is an instance of $e_1\sigma_1$, i.e., there exists a substitution θ such that $e_2\sigma_2 = e_1\sigma_1\theta$.

Unfortunately, given the infinite computation shown in Example 6:

$$\begin{aligned}
\langle \mathsf{add}(X, Y), id\rangle &\overset{r_4}{\leadsto} \langle \mathsf{S}(\mathsf{add}(X', Y)), \{X \mapsto \mathsf{S}(X')\}\rangle\\
&\overset{r_4}{\leadsto} \langle \mathsf{S}(\mathsf{S}(\mathsf{add}(X', Y))), \{X \mapsto \mathsf{S}(\mathsf{S}(X'')), X' \mapsto \mathsf{S}(X'')\}\rangle\\
&\overset{r_4}{\leadsto} \ldots
\end{aligned}$$

subsumption does not suffice to stop this derivation. Basically, new states do not subsume previous states because of the increasing number of topmost constructors S. This is usually solved in functional languages either by introducing further operations for removing the topmost constructor symbols from the states or by only considering function calls (i.e., maximal subexpressions rooted by defined function symbols). Here, we prefer to keep the framework as simple as possible, so this situation will be dealt with our second operator: abstraction (see below).

3.2. Abstraction

While subsumption allows one to produce finite-state symbolic execution graphs in many cases, this cannot be always ensured. In some cases, some form of *abstraction* is also required. For this purpose, we first introduce the notion of *composition* operator. In the following, we let $\Longrightarrow$ denote the multiset extension of the symbolic execution transition relation $\leadsto$, i.e., given a multiset of symbolic states $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$, we have $(\mathcal{S}_1, \ldots, \mathcal{S}_i, \ldots, \mathcal{S}_n) \overset{\tau,\pi}{\Longrightarrow} (\mathcal{S}_1, \ldots, \mathcal{S}'_i, \ldots, \mathcal{S}_n)$ iff $\mathcal{S}_i \overset{\tau,\pi}{\leadsto} \mathcal{S}'_i$ for some $i \in \{1, \ldots, n\}$.

Definition 8. Let $\langle \Sigma, \Theta, \mathcal{T}, \rho\rangle$ be a program and $\langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}^\sharp, \rho^\sharp\rangle$ its symbolic version. We say that the (associative and commutative) binary operator $\oplus$ is a *composition* operator if the following conditions hold:

- $\mathcal{S}_i \sqsubseteq_\gamma \mathcal{S}'_i$ implies $\mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}_i \oplus \ldots \oplus \mathcal{S}_n \sqsubseteq_\gamma \mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}'_i \oplus \ldots \oplus \mathcal{S}_n$, $1 \leq i \leq n$, with $\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{S}'_i \in \Sigma^\sharp$.
- If $\mathcal{S} \sqsubseteq_\gamma \mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}_n$ and there is an infinite computation from some $s \in \gamma(\mathcal{S})$, then there is also an infinite computation from some $s' \in \gamma(\mathcal{S}_i)$, $i \in \{1, \ldots, n\}$.

Roughly speaking, the first condition ensures that generalizing part of an expression also generalizes the complete expression. The second condition is needed to ensure that analyzing the termination of the computations from $\mathcal{S}_1, \ldots, \mathcal{S}_n$ independently suffices for inferring the termination of the computations from $\mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}_n$.

Definition 9 *(Abstraction operator).* Let $\mathcal{S}$ be a symbolic state and let $\mathcal{C}$ be a set of symbolic states (typically, a set of previous symbolic states). We say that $\alpha : \Sigma^\sharp \times \wp(\Sigma^\sharp) \mapsto \wp(\Sigma^\sharp)$ is an *abstraction operator* if $\alpha(\mathcal{S}, \mathcal{C}) = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, $n > 0$, implies $\mathcal{S} \sqsubseteq_\gamma \mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}_n$ for some composition operator $\oplus$.

Loosely speaking, an abstraction operator may *decompose* a symbolic state into a number of symbolic states whose composition (using an appropriate operator $\oplus$) is more general than the original symbolic state. This *divide and conquer* strategy is often essential to obtain a finite graph. The abstraction operator usually takes into account the computation history (i.e., the previous states in the same computation). As mentioned before, since the ascending chain condition holds for $\sqsubseteq_\gamma$, abstraction suffices to eventually build a graph where all states can be folded back using subsumption, thus ensuring the construction of a *finite* graph.

Definition 10 *(Abstraction step).* Let $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}, \rho^\sharp \rangle$ be a symbolic program and $\mathcal{G}_{P^\sharp}$ be an incomplete symbolic execution graph for $\mathcal{S}_0 \in \Theta^\sharp$ where

$$\underline{\mathcal{S}_0} \longrightarrow \underline{\mathcal{S}_1} \longrightarrow \ldots \longrightarrow \underline{\mathcal{S}_{n-1}} \longrightarrow \mathcal{S}_n$$

is a path in the graph with $n > 0$, so that all nodes are marked but the last one. Given an abstraction operator α, we perform an abstraction step by marking $\mathcal{S}_n$ and adding edges labeled with abs from $\mathcal{S}_n$ to new nodes labeled with the symbolic states in $\alpha(\mathcal{S}_n, \mathcal{C})$, with $\mathcal{C} \subseteq \{\mathcal{S}_0, \ldots, \mathcal{S}_{n-1}\}$.

We note that, in contrast to our approach, abstraction in symbolic execution is typically used to *underapproximate* the computation space.[4]

Now, we present a simple and generic strategy to apply abstraction while ensuring the finiteness of the symbolic execution graph. For this purpose, we first recall the notion of well-quasi-order:

Definition 11 *(Well-quasi-ordering, wqo).* A well-quasi-ordering $\leq$ on a set S is a quasi-ordering (i.e., a reflexive and transitive binary relation) such that any infinite sequence of elements $e_1, e_2, e_3, \ldots$ from S contains an increasing pair $e_i \leq e_j$ with $i < j$. The set $(S, \leq)$ is said to be well-quasi-ordered.

Here, we plan to use a wqo $(\Sigma^\sharp, \preceq)$ on symbolic states so that only states in the same equivalence class w.r.t. a comparison order $\sim$ are comparable with the ordering $\preceq$ (which is safe by Ramsey's theorem [41], since the number of equivalence classes is finite).

Summarizing, we can use the following algorithm to construct a finite symbolic execution graph:

Definition 12 *(Finite graph construction).* Let $P^\sharp = \langle \Sigma^\sharp, \Theta^\sharp, \mathcal{T}, \rho^\sharp \rangle$ be a symbolic program and $\mathcal{G}_{P^\sharp}$ be an incomplete symbolic execution graph for $\mathcal{S}_0 \in \Theta^\sharp$. We consider that $\sim$ is a comparison relation and $(\Sigma^\sharp, \preceq)$ is a wqo.

```
**Initialization:**
    $i := 0$;  $T_i$ is a tree with one *unmarked* node labeled with $\mathcal{S}_0$.
**Repeat**
    1. Let $\underline{\mathcal{S}_0} \longrightarrow \underline{\mathcal{S}_1} \longrightarrow \ldots \longrightarrow \underline{\mathcal{S}_{n-1}} \longrightarrow \mathcal{S}_n$ be a root-to-leaf path in $T_i$.
    2. We consider three possibilities to expand the graph:[5]
      • If $\mathcal{S}_n$ subsumes a comparable symbolic state $\mathcal{S}_i$, $0 \leq i \leq n-1$, with $\mathcal{S}_n \sim \mathcal{S}_i$, $T_{i+1}$ is obtained by adding a
        subsumption edge from $\mathcal{S}_n$ to $\mathcal{S}_i$ and marking the node $\mathcal{S}_n$.
      • Otherwise, we check if $\mathcal{S}_n$ is bigger than some comparable symbolic state $\mathcal{S}_i$, $0 \leq i \leq n-1$, i.e., $\mathcal{S}_i \preceq \mathcal{S}_n$ and $\mathcal{S}_n \sim$
        $\mathcal{S}_i$. In this case, $T_{i+1}$ is obtained by applying an abstraction step to $\mathcal{S}_n$ w.r.t. the set of comparable ancestors,
        and marking this node.
      • Otherwise, one symbolic execution step is performed. Here, $T_{i+1}$ is obtained by marking $\mathcal{S}_n$ and adding the
        corresponding children (which can be none if the state is a final state).
    3. $i := i + 1$
**until** $T_i$ does not contain unmarked nodes
```

Finiteness of the symbolic execution graph is an easy consequence of the following facts:

- Since $(\Sigma^\sharp, \preceq)$ is a wqo, by Definition 11, any infinite path in the symbolic execution graph $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \ldots$ must contains an increasing pair $\mathcal{S}_i \preceq \mathcal{S}_j$ with $i < j$. Therefore, abstraction is applied to these paths.
- Moreover, every abstraction step reduces a symbolic state $\mathcal{S}$ to a (finite) set of new abstracted states $\mathcal{S}_1, \ldots, \mathcal{S}_n$ such that $\mathcal{S} \sqsubseteq_\gamma \mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}_n$. Furthermore, by Definition 8, $\mathcal{S}_i \sqsubseteq_\gamma \mathcal{S}'_i$ implies $\mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}_i \oplus \ldots \oplus \mathcal{S}_n \sqsubseteq \mathcal{S}_1 \oplus \ldots \oplus \mathcal{S}'_i \oplus \ldots \oplus \mathcal{S}_n$, so further abstractions to an element of a composition still generalizes the composed expression. Finally, since $(\Sigma^\sharp, \sqsubseteq_\gamma)$ satisfies the ascending chain condition, a symbolic state can only be abstracted a finite number of times, so that subsumption eventually applies.

[4] Actually, [6] already suggests in the conclusion how to compute an *overapproximation* by also evaluating abstracted states, as we do.

[5] Here, we consider that the comparable symbolic states are tried sequentially from the root of the graph to the current node.

Example 9. Let us first consider the language **IMP**. Here, for simplicity, we assume that integer variables are bounded between $-minint$ and $+maxint$. Then, we can easily define a wqo $(\Sigma^\sharp, \preceq_{\mathsf{imp}})$ on comparable symbolic states as follows:

$$\begin{array}{l}\langle l, \sigma_1, \pi_1\rangle \preceq_{\mathsf{imp}} \langle l, \sigma_2, \pi_2\rangle \\ \quad \text{iff } \{\overline{x\sigma_1\theta_1} \mid x \in V,\ \theta_1 \in \mathsf{sols}_V(\pi_1)\} \subseteq \{\overline{x\sigma_2\theta_2} \mid x \in V,\ \theta_2 \in \mathsf{sols}_V(\pi_2)\}\end{array}$$

where V is the set of the program variables and $\bar{e}$ denotes the evaluation of expression e. For instance, for the program WHILE$^\sharp$, the biggest state has the form $\{l, \{x \mapsto X\}, true\}$, since it denotes the set $\{-minint, -minint + 1, \ldots, 0, \ldots, +maxint - 1, +maxint\}$. Alternatively, one could consider an infinite integer domain and represent symbolic values with integer intervals, so that abstraction is based on some widening operator, as discussed in page 146.

In our previous example, it is not possible to construct a symbolic execution where some state is bigger (using $\preceq_{\mathsf{imp}}$) than a previous state since variables can only become more constrained. However, we can slightly modify the program as follows:

$$\begin{array}{ll} l_0: & x := 1; \\ l_1: & \textbf{while } x > 0 \textbf{ do} \\ l_2: & \quad x := \textbf{input}(); \\ l_3: & \textbf{done} \end{array}$$

$$\begin{tikzcd} l_0 \arrow[r, "\tau_1"] & l_1 \arrow[r, bend left, "\tau_2"] \arrow[d, "\tau_4"'] & l_2 \arrow[l, bend left, "\tau_3"] \\ & l_3 & \end{tikzcd}$$

Now, we have the following symbolic execution:

$$\begin{array}{rl}\langle l_0, \{x \mapsto X\}, true\rangle & \overset{\tau_1}{\leadsto} \langle l_1, \{x \mapsto 1\}, true\rangle \\ & \overset{\tau_2}{\leadsto} \langle l_2, \{x \mapsto 1\}, true\rangle \\ & \overset{\tau_3}{\leadsto} \langle l_1, \{x \mapsto X'\}, true\rangle\end{array}$$

so that $\langle l_1, \{x \mapsto X'\}, true\rangle$ is not subsumed by the previous comparable state $\langle l_1, \{x \mapsto 1\}, true\rangle$ but we have

$$\langle l_1, \{x \mapsto 1\}, true\rangle \preceq_{\mathsf{imp}} \langle l_1, \{x \mapsto X'\}, true\rangle$$

since $\{1\} \subseteq \{-minint, \ldots, 0, \ldots, +maxint\}$. In our simple language **IMP**, no composition operator is required since abstraction would only return a single, more general symbolic state. In the interprocedural case, however, states usually contain a stack and abstraction may need to produce more than one new symbolic state.

Here, we can use a simple abstraction operator that simply assigns a fresh symbolic variable to each program variable that keeps growing. Therefore, the symbolic execution proceeds as follows:

$$\begin{array}{rl}\langle l_1, \{x \mapsto X'\}, true\rangle & \overset{\mathsf{abs}}{\leadsto} \langle l_1, \{x \mapsto X''\}, true\rangle \equiv \mathcal{S}_5 \\ & \overset{\tau_2}{\leadsto} \langle l_2, \{x \mapsto X''\}, X'' > 0\rangle \\ & \overset{\tau_3}{\leadsto} \langle l_1, \{x \mapsto X'''\}, X'' > 0\rangle \\ & \overset{\mathsf{sub}}{\leadsto} \mathcal{S}_5\end{array}$$

and the symbolic execution is thus finite now.

Example 10. In the context of logic and functional programming, the definition of a wqo is often based on some variant of the *homeomorphic embedding* ordering [27,19]. Intuitively speaking, $e_1 \preceq_{\mathsf{func}} e_2$ if e_1 can be obtained from e_2 by deleting some function symbols (see, e.g., [29] for a precise definition). E.g., $\mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{b})), \mathsf{b})$ embeds (is bigger than) $\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{b}))$.

Let us consider again the infinite symbolic execution of Example 8:

$$\begin{array}{rl}\langle \mathsf{add}(X, Y), id\rangle & \overset{r_4}{\leadsto} \langle \mathsf{S}(\mathsf{add}(X', Y)), \{X \mapsto \mathsf{S}(X')\}\rangle \\ & \overset{r_4}{\leadsto} \langle \mathsf{S}(\mathsf{S}(\mathsf{add}(X', Y))), \{X \mapsto \mathsf{S}(\mathsf{S}(X'')), X' \mapsto \mathsf{S}(X'')\}\rangle \\ & \overset{r_4}{\leadsto} \ldots\end{array}$$

Here, we consider that $\langle e_1, \sigma_1\rangle \preceq_{\mathsf{func}} \langle e_2, \sigma_2\rangle$ iff the states are comparable and $e_1 \preceq_{\mathsf{func}} e_2$. Then, we have that

$$\langle \mathsf{S}(\mathsf{add}(X', Y)), \{X \mapsto \mathsf{S}(X')\}\rangle \preceq_{\mathsf{func}} \langle \mathsf{S}(\mathsf{S}(\mathsf{add}(X', Y))), \{X \mapsto \mathsf{S}(\mathsf{S}(X'')), X' \mapsto \mathsf{S}(X'')\}\rangle$$

and they are comparable since both states have S as the root symbol.

Roughly speaking, in our functional context, abstraction may take an expression containing nested function symbols, e.g., $\mathsf{f}(\mathsf{g}(X), Y)$, and return new states containing the constituents of this expression, e.g., $\mathsf{f}(W, Y)$ and $W = \mathsf{g}(X)$, where W is a fresh symbolic variable, so that $\mathsf{f}(W, Y) \oplus (W = \mathsf{g}(X)) = \mathsf{f}(\mathsf{g}(X), Y)$. For simplicity, we do not make the link W explicit (since it is not relevant for our purposes: proving termination) and just write $\mathsf{f}(W, Y)$ and $\mathsf{g}(X)$.

In particular, an abstraction operator can be naturally defined using the notion of *least general generalization* [37]. First, given two expressions, e_1 and e_2, we say that e is a generalization of e_1 and e_2 if there exist substitutions θ_1 and θ_2 such that $e\theta_1 = e_1$ and $e\theta_2 = e_2$; we say that e is the least general generalization (*lgg*) if e is an instance of any other generalization. We assume a function *lgg* such that $lgg(e_1, e_2) = (e, \theta_1, \theta_2)$. For instance, $lgg(\mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{b}),\mathsf{f}(\mathsf{g}(\mathsf{b}),\mathsf{c}))= (\mathsf{f}(\mathsf{g}(X),Y), \{X\mapsto \mathsf{a}, Y\mapsto \mathsf{b}\}, \{X\mapsto \mathsf{b}, Y\mapsto \mathsf{c}\})$.

Essentially, given two symbolic states $s_1 = \langle e_1, \sigma_1\rangle$ and $s_2 = \langle e_2, \sigma_2\rangle$ with $lgg(e_1, e_2) = (e, \theta_1, \theta_2)$, the abstraction of s_2 w.r.t. s_1 returns the symbolic state $\langle e, \sigma_2\rangle$, together with new states $\langle e', \sigma_2\rangle$ for each expression containing defined functions in θ_2.

Here, the *lgg* of $\mathsf{S}(\mathsf{add}(X', Y))$ and $\mathsf{S}(\mathsf{S}(\mathsf{add}(X', Y)))$ returns

$$(\mathsf{S}(W),\ \{W \mapsto \mathsf{add}(X', Y)\},\ \{W \mapsto \mathsf{S}(\mathsf{add}(X', Y))\})$$

Therefore, an abstraction step proceeds as follows:

$$\begin{array}{l}\langle \mathsf{S}(\mathsf{S}(\mathsf{add}(X', Y))), \{X \mapsto \mathsf{S}(\mathsf{S}(X'')), X' \mapsto \mathsf{S}(X'')\}\rangle\\ \quad\overset{\mathsf{abs}}{\leadsto}\ \langle \mathsf{S}(\mathsf{add}(X', Y)), \{X \mapsto \mathsf{S}(\mathsf{S}(X'')), X' \mapsto \mathsf{S}(X'')\}\rangle\end{array}$$

and the new symbolic state subsumes a previous state, so the computation terminates.

In other cases, abstraction returns more than one relevant (i.e., with defined functions) symbolic states and we have to follow several paths, e.g.,

$$\begin{array}{l}\langle \mathsf{main}(X, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), id\rangle\\ \quad\overset{r_0}{\leadsto}\ \langle \mathsf{mult}(X, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), id\rangle\\ \quad\overset{r_2}{\leadsto}\ \langle \mathsf{add}(\mathsf{mult}(W, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \{X \mapsto \mathsf{S}(W)\}\rangle\\ \quad\overset{r_2}{\leadsto}\ \langle \mathsf{add}(\mathsf{add}(\mathsf{mult}(W', \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \{X \mapsto \mathsf{S}(\mathsf{S}(W')), W \mapsto \mathsf{S}(W')\}\rangle \equiv \mathcal{S}\end{array}$$

with

$$\mathcal{S} \overset{abs}{\leadsto} \langle \mathsf{add}(X', \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \{X \mapsto \mathsf{S}(\mathsf{S}(W')), W \mapsto \mathsf{S}(W')\}\rangle$$

and

$$\mathcal{S} \overset{abs}{\leadsto} \langle \mathsf{mult}(W, \mathsf{S}(\mathsf{S}(\mathsf{Z}))), \{X \mapsto \mathsf{S}(\mathsf{S}(W')), W \mapsto \mathsf{S}(W')\}\rangle$$

The second symbolic state is already subsumed by a previous one, while the first one will require a few further steps.

The complete symbolic execution graph for program $\mathsf{MULT}^\sharp$ is shown in Fig. 5, where we skip the substitutions of the symbolic states to keep the graph simple.

We have presented the conditions that an abstraction strategy must fulfill in order to ensure the finiteness of the symbolic execution graph and its correctness (i.e., that an overapproximation is still computed, see Theorem 1 below). Of course, the definition of a good abstraction strategy depends on the considered programming language, since the definition of both the wqo $\preceq$ and the abstraction operator α highly depend on the features of the programming language. We refer the interested reader to papers dealing with concrete programming languages (like, e.g., Java [10], where an abstraction operation is introduced in order to merge two symbolic states whenever the evaluation reaches a program position for the second time).

3.3. Closed symbolic execution graphs

In the previous section, we have introduced subsumption and abstraction operators that allow one (using appropriate strategies) to construct a finite symbolic execution graph. However, is the resulting graph still complete? (i.e., does it still represent an overapproximation of the original computations?). In this section, we introduce the notion of *closed* symbolic execution graph,[6] which suffices for completeness.

Definition 13 *(Closed symbolic execution graph).* Let P be a program and $P^\sharp$ a symbolic version of P. Let $\mathcal{G}$ be a *finite* symbolic execution graph (possibly including subsumption and abstraction steps). Then, $\mathcal{G}$ is closed if all nodes are marked.

The closedness of a symbolic execution graph guarantees that all concrete executions are covered in the graph (i.e., that it computes an overapproximation). We note that our closed symbolic execution graphs have some similarities with the *abstract reachability graphs* of [25]; however, the abstract reachability graph only represents an overapproximation of concrete execution when the graph is finite, which is not always ensured (though some strategies are discussed).

Finally, we present the main results of this section, which show that closed symbolic execution graphs indeed represent an overapproximation of concrete execution and can be used for proving the termination of the original program.

[6] The terminology "closed" is taken from the partial evaluation literature.

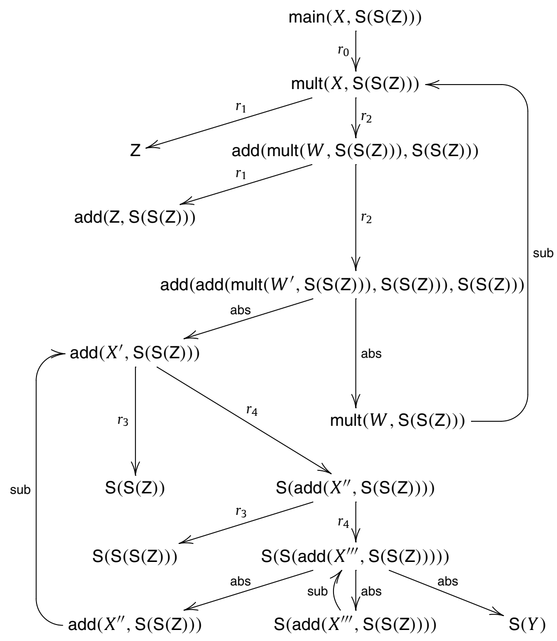


Fig. 5. Symbolic execution graph for program MULT$^\sharp$.

Theorem 1. *Let P be a program and $P^\sharp$ a symbolic version of P. Let $\mathcal{G}$ be a closed symbolic execution graph for $\mathcal{S}_0$ and let $s_0 \in \gamma(\mathcal{S}_0)$. For every computation $s_0 \overset{\tau_1}{\to}_{R_P} s_1 \overset{\tau_2}{\to}_{R_P} \ldots$, there is a symbolic execution $\mathcal{S}_0 \overset{\tau_1}{\leadsto}_{R_{P^\sharp}} \mathcal{S}_1 \overset{\tau_2}{\leadsto}_{R_{P^\sharp}} \ldots$ such that for all $\mathcal{S}_i \overset{\tau_{i+1}}{\leadsto} \mathcal{S}_{i+1}$, $i \geq 0$, either the edge $\mathcal{S}_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$ belongs to the graph $\mathcal{G}$ or there are nodes $\mathcal{S}_{i1}, \ldots, \mathcal{S}_{in}$, $n > 1$, in $\mathcal{G}$ with $\mathcal{S}_i = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{ij} \oplus \ldots \oplus \mathcal{S}_{in}$, $\mathcal{S}_{i+1} = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}'_{ij} \oplus \ldots \oplus \mathcal{S}_{in}$, $1 \leq j \leq n$, and the edge $\mathcal{S}_{ij} \xrightarrow{\tau_{i+1}} \mathcal{S}'_{ij}$ belongs to $\mathcal{G}$.*

Proof. We prove the claim by induction. Since the base case is trivial, we consider the inductive case. Consider an arbitrary transition $s_i \overset{\tau_{i+1}}{\to}_{R_P} s_{i+1}$, $i > 0$. By condition (4) of Definition 4, we have that, for any $\mathcal{S}_i$ such that $s_i \in \gamma(\mathcal{S}_i)$, the transition $\mathcal{S}_i \overset{\tau_{i+1}}{\leadsto}_{R_P^\sharp} \mathcal{S}_{i+1}$ holds with $s_{i+1} \in \gamma(\mathcal{S}_{i+1})$. First, we consider that the graph contains a node labeled with $\mathcal{S}_i$. Since the graph is closed, we only need to distinguish the following cases[7]:

- The graph has an edge $\mathcal{S}_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$. Then, the claim follows by induction.
- The graph contains an edge $\mathcal{S}_i \xrightarrow{\text{sub}} \mathcal{S}'_i$. By Definition 6, $\mathcal{S}_i \sqsubseteq_\gamma \mathcal{S}'_i$ and, thus, $\gamma(\mathcal{S}_i) \subseteq \gamma(\mathcal{S}'_i)$. Therefore, we have $s_i \in \gamma(\mathcal{S}'_i)$ too. By condition (4) of Definition 4, the transition $\mathcal{S}'_i \overset{\tau_{i+1}}{\leadsto}_{R_P^\sharp} \mathcal{S}_{i+1}$ holds too with $s_{i+1} \in \gamma(\mathcal{S}_{i+1})$. Hence, $\mathcal{S}'_i \xrightarrow{\tau_{i+1}} \mathcal{S}_{i+1}$ belongs to the graph and the proof follows by induction.
- The graph contains edges $S_i \xrightarrow{\text{abs}} \mathcal{S}_{i1}, \ldots, S_i \xrightarrow{\text{abs}} \mathcal{S}_{in}$, $n \geq 1$. By Definitions 9 and 10, we have $\mathcal{S}_i \sqsubseteq_\gamma \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{in}$ and, thus, $\gamma(\mathcal{S}_i) \subseteq \gamma(\mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{in})$. Therefore, we have $s_i \in \gamma(\mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{in})$. Let $\mathcal{S}' = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{in}$. By condition (4) of Definition 4, the transition $\mathcal{S}' \overset{\tau_{i+1}}{\leadsto}_{R_P^\sharp} \mathcal{S}''$ holds too with $s_{i+1} \in \gamma(\mathcal{S}'')$. By Definition, $\mathcal{S}' \overset{\tau_{i+1}}{\leadsto} \mathcal{S}''$ implies $(\mathcal{S}_{i1}, \ldots, \mathcal{S}_{ij}, \ldots, \mathcal{S}_{in}) \overset{\tau_{i+1}}{\Longrightarrow} (\mathcal{S}_{i1}, \ldots, \mathcal{S}'_{ij}, \ldots, \mathcal{S}_{in})$, $1 \leq j \leq n$, with $\mathcal{S}'' = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}'_{ij} \oplus \ldots \oplus \mathcal{S}_{in}$. Therefore, we have $\mathcal{S}_{ij} \overset{\tau_{i+1}}{\leadsto} \mathcal{S}'_{ij}$. For simplicity, we assume that there are no consecutive abstraction or subsumption steps (though extending the proof for the general case is not difficult). Thus, we have $\mathcal{S}_{ij} \xrightarrow{\tau_{i+1}} \mathcal{S}'_{ij}$ in the graph, and the proof follows by induction.

Let us now consider that the graph contains nodes labeled with $\mathcal{S}_{i1}, \ldots, \mathcal{S}_{in}$ such that $\mathcal{S}_i = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{in}$. Then, we distinguish the following possibilities:

[7] We note that if the graph were not closed, one should also consider (unmarked) nodes without output edges, and the claim would not be true.

- The graph has an edge $\mathcal{S}_{ij} \xrightarrow{\tau_{i+1}} \mathcal{S}'_{ij}$, $1 \leq j \leq n$, with $\mathcal{S}_{i+1} = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}'_{ij} \oplus \ldots \oplus \mathcal{S}_{in}$, and the claim follows by induction.
- The graph contains an edge $\mathcal{S}_{ij} \xrightarrow{\mathsf{sub}} \mathcal{S}'_{ij}$, $1 \leq j \leq n$. By Definition 6, we have $\mathcal{S}_{ij} \sqsubseteq_\gamma \mathcal{S}'_{ij}$. Now, by Definition 8, we have $\mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{ij} \oplus \ldots \oplus \mathcal{S}_{in} \sqsubseteq_\gamma \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}'_{ij} \oplus \ldots \oplus \mathcal{S}_{in}$. Let $\mathcal{S}' = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}'_{ij} \oplus \ldots \oplus \mathcal{S}_{in}$. Then, we have $s_i \in \gamma(\mathcal{S}')$ too. By condition (4) of Definition 4, the transition $\mathcal{S}' \overset{\tau_{i+1}}{\leadsto}_{R_P^\sharp} \mathcal{S}''$ holds too with $s_{i+1} \in \gamma(\mathcal{S}'')$. Hence, $\mathcal{S}_a \xrightarrow{\tau_{i+1}} \mathcal{S}_b$ belongs to the graph, with $\mathcal{S}_a = \mathcal{S}_{ik}$, $\mathcal{S}_b = \mathcal{S}'_{ik}$, $k \in \{1, \ldots, j-1, j+1, \ldots, n\}$ and $\mathcal{S}'' = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}'_{ik} \oplus \ldots \oplus \mathcal{S}_{in}$, or $\mathcal{S}_a = \mathcal{S}'_{ij}$, $\mathcal{S}_b = \mathcal{S}''_{ij}$, and $\mathcal{S}'' = \mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}''_{ij} \oplus \ldots \oplus \mathcal{S}_{in}$. In either case, the proof follows by induction.
- The graph contains edges $S_{ij} \xrightarrow{\mathsf{abs}} \mathcal{S}'_1, \ldots, S_{ij} \xrightarrow{\mathsf{abs}} \mathcal{S}'_m$, $m \geq 1$, $1 \leq j \leq n$. By Definition 9, we have $\mathcal{S}_{ij} \sqsubseteq_\gamma \mathcal{S}'_1 \oplus \ldots \oplus \mathcal{S}'_m$ and, by Definition 8, $\mathcal{S}_{i1} \oplus \ldots \oplus \mathcal{S}_{ij} \oplus \ldots \oplus \mathcal{S}_{in} \sqsubseteq_\gamma \mathcal{S}_{i1} \oplus \ldots \oplus (\mathcal{S}'_1 \oplus \ldots \oplus \mathcal{S}'_m) \oplus \ldots \oplus \mathcal{S}_{in} = \mathcal{S}'$. Therefore, we have $s_i \in \gamma(\mathcal{S}')$. By condition (4) of Definition 4, the transition $\mathcal{S}' \overset{\tau_{i+1}}{\leadsto}_{R_P^\sharp} \mathcal{S}''$ holds too with $s_{i+1} \in \gamma(\mathcal{S}'')$. By definition, $\mathcal{S}' \overset{\tau_{i+1}}{\leadsto} \mathcal{S}''$ implies $(\mathcal{S}_{i1}, \ldots, \mathcal{S}'_1, \ldots, \mathcal{S}'_m, \ldots, \mathcal{S}_{in}) = (\mathcal{S}''_1, \ldots, \mathcal{S}''_l, \ldots, \mathcal{S}''_k) \xRightarrow{\tau_{i+1}} (\mathcal{S}''_1, \ldots, \mathcal{S}'''_l, \ldots, \mathcal{S}''_k)$, $1 \leq l \leq k$, $k = n + m - 1$, and the proof proceeds as in the previous case. □

Thanks to Theorem 1 and the conditions of Definition 8, one can analyze the termination of the original program by analyzing the transitions (which are not subsumption or abstraction steps) in the closed symbolic execution graph. Actually, one can further restrict it to the transitions that belong to a loop in the graph (i.e., to the strongly connected components of the graph).

Our approach is clearly designed to perform a termination analysis. This is why the previous result, Theorem 1, only guarantees that there is a symbolic counterpart for every concrete execution. Nevertheless, one could extend the framework to also preserve other properties of interest so that other kind of analyses would be possible.

3.4. The approach in practice

In this paper, we have introduced two simple programming languages to illustrate our developments. Furthermore, in [47] we presented a termination prover for the language IMP. This tool constructs a finite symbolic execution graph and, then, extracts rewrite rules from the transitions in the graph. Finally, termination of the rewrite rules is analyzed using the termination prover AProVE [22]. A web interface to test the tool is available from

http://kaz.dsic.upv.es/sett/

Clearly, both IMP and FUNC are tiny languages, still far from real world programming languages. However, as mentioned in the introduction, there are a number of powerful tools that have been already developed and that mostly follow the scheme presented in this paper. For instance, the system AProVE [22] for proving the termination of Prolog, Haskell and Java (as well as Costa [3,2] and Julia [43] to some extent, see below), follow a similar scheme and have been applied to real world programming languages. Therefore, the viability of the approach should be clear.

4. Related work

As mentioned before, there are already several approaches to proving the termination of programs which mostly follow a similar scheme as the one we have presented. This is the case, e.g., of the works that consider the termination of Haskell [21], Prolog with impure features [42], narrowing [35,46], and Java bytecode [36,10,9] by transforming the original termination problem into the problem of analyzing the termination of a rewrite system. Costa [3,2], a cost and termination analyzer for Java bytecode, follows a similar pattern but produces a constraint logic program instead. Moreover, in contrast to our approach, Costa basically uses a sort of control flow graph of the program—rather than a symbolic execution graph—to guide the compilation of a Java program into a constraint logic program. Julia [43] is also a termination prover for Java that follows a similar scheme as Costa but considers the result of a so called *path-length analysis* to compute the associated constraint logic programs.

The novelty of our approach is twofold. On the one hand, we propose a language-independent approach that may ease the design of new program analyzers for different programming languages by clarifying some common principles of these approaches. On the other hand, we reformulate the scheme using well-known principles from symbolic execution and partial evaluation, so that the vast literature on constructing finite symbolic executions can be reused (rather than starting from scratch, as some of the above works have done).

Proving that a program terminates for all possible inputs is undoubtedly a fundamental problem that has been extensively studied in the context of term rewriting [8,45] and logic programming [31], where powerful termination provers exist (see, e.g., the results from the last *termination competition* [1]). In contrast, proving the termination of imperative programs has been mostly overlooked for decades. Recent progress in this area, however, has changed the picture and powerful—and usable—tools have emerged [44].

A popular recent branch of work is based on the notion of *transition invariants* [39] and applies to both sequential and concurrent programs (see [14] for a recent survey). This technique aims at identifying a set of invariants that approximate the closure of the transition relation of a program, so that if these invariants are well founded, the considered program is terminating. The main advantage of this method is that his divide-and-conquer approach allows one to search for different well-founded relations rather than a single, monolithic one for the complete program (which is much more difficult in practice). This method, however, relies on the construction of ranking functions and, thus, our symbolic execution-based approach may be advantageous when the control flow is complex (but can be represented with a finite number of states without losing too much precision). Actually, our preliminary experimental results showed that our scheme succeeds for some typical examples from the transition invariants literature [47].

Another alternative approach considers the termination of C programs by translating the original program to a term rewrite system [20]. However, in contrast to our approach, the rewrite rules are directly extracted from the program's syntax. Consequently, it is (faster but) less accurate since no information is propagated forward in the computations. In order to alleviate this problem, additional static analyses are proposed, though their impact is difficult to measure.

5. Conclusion

In this work, we have presented the front-end of a language-independent approach to proving program termination by constructing a finite and complete symbolic execution graph. We have illustrated our approach using two simple imperative and functional programming languages. Hopefully, this higher level approach will provide useful insights for designing new semantics-based termination tools for particular programming languages.

References

[1] Annual international termination competition, http://www.termination-portal.org/wiki/Termination_Competition.

[2] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, D. Zanardini, Termination analysis of Java bytecode, in: G. Barthe, F.S. de Boer (Eds.), Proc. of FMOODS'08, in: Lecture Notes in Computer Science, vol. 5051, Springer, 2008, pp. 2–18.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, COSTA: design and implementation of a cost and termination analyzer for Java bytecode, in: Proc. of FMCO'07, in: Lecture Notes in Computer Science, vol. 5382, Springer, 2008, pp. 113–132.

[4] E. Albert, G. Vidal, The narrowing-driven approach to functional logic program specialization, New Gener. Comput. 20 (1) (2002) 3–26.

[5] S. Anand, C.S. Pasareanu, W. Visser, Symbolic execution with abstract subsumption checking, in: A. Valmari (Ed.), Proc. of SPIN'06, in: Lecture Notes in Computer Science, vol. 3925, Springer, 2006, pp. 163–181.

[6] S. Anand, C.S. Pasareanu, W. Visser, Symbolic execution with abstraction, Int. J. Softw. Tools Technol. Transf. 11 (1) (2009) 53–67.

[7] T. Arts, J. Giesl, Termination of term rewriting using dependency pairs, Theor. Comput. Sci. 236 (1–2) (2000) 133–178.

[8] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[9] M. Brockschmidt, C. Otto, J. Giesl, Modular termination proofs of recursive Java bytecode programs by term rewriting, in: Proc. of RTA 2011, in: LIPIcs, vol. 10, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2011, pp. 155–170.

[10] M. Brockschmidt, C. Otto, C. von Essen, J. Giesl, Termination graphs for Java bytecode, in: S. Siegler, N. Wasser (Eds.), Verification, Induction, Termination Analysis, in: Lecture Notes in Computer Science, vol. 6463, Springer, 2010, pp. 17–37.

[11] L. Clarke, A program testing system, in: Proceedings of the 1976 Annual Conference (ACM'76), 1976, pp. 488–491.

[12] M. Colón, H. Sipma, Synthesis of linear ranking functions, in: T. Margaria, W. Yi (Eds.), Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001, Genova, Italy, April 2–6, 2001, in: Lecture Notes on Computer Science, vol. 2031, Springer, 2001, pp. 67–81, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, 2001.

[13] M. Colón, H. Sipma, Practical methods for proving program termination, in: E. Brinksma, K. Larsen (Eds.), Proceedings of the 14th International Conference on Computer Aided Verification, CAV 2002, Copenhagen, Denmark, July 27–31, 2002, in: Lecture Notes on Computer Science, vol. 2404, Springer, 2004, pp. 442–454.

[14] B. Cook, A. Podelski, A. Rybalchenko, Proving program termination, Commun. ACM 54 (5) (2011) 88–98.

[15] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. of Fourth ACM Symp. on Principles of Programming Languages, 1977, pp. 238–252.

[16] P. Cousot, R. Cousot, Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper, in: M. Bruynooghe, M. Wirsing (Eds.), Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP'92, Leuven, Belgium, 13–17 August 1992, in: Lecture Notes in Computer Science, vol. 631, Springer-Verlag, Berlin, Germany, 1992, pp. 269–295.

[17] D. De Schreye, S. Decorte, Termination of logic programs: the never-ending story, J. Log. Program. 19/20 (1994) 199–260.

[18] N. Dershowitz, Termination of rewriting, J. Symb. Comput. 3 (1&2) (1987) 69–115.

[19] N. Dershowitz, J.-P. Jouannaud, Rewrite systems, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics, Elsevier, Amsterdam, 1990, pp. 243–320.

[20] S. Falke, D. Kapur, C. Sinz, Termination analysis of C programs using compiler intermediate languages, in: M. Schmidt-Schauß (Ed.), Proc. of RTA'11, in: LIPIcs, vol. 10, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2011, pp. 41–50.

[21] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, R. Thiemann, Automated termination proofs for Haskell by term rewriting, ACM Trans. Program. Lang. Syst. 33 (2) (2011) 7.

[22] J. Giesl, P. Schneider-Kamp, R. Thiemann, AProVE 1.2: automatic termination proofs in the dependency pair framework, in: Proc. of IJCAR'06, in: Lecture Notes on Computer Science, vol. 4130, Springer, 2006, pp. 281–286.

[23] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, C. Fuhs, Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs, in: PPDP'12, ACM, 2012, pp. 1–12.

[24] M. Heizmann, J. Hoenicke, J. Leike, A. Podelski, Linear ranking for linear lasso programs, in: D.V. Hung, M. Ogawa (Eds.), Proceedings of 11th International Symposium on Automated Technology for Verification and Analysis, ATVA 2013, Hanoi, Vietnam, October 15–18, 2013, in: Lecture Notes on Computer Science, vol. 8172, Springer, 2013, pp. 365–380.

[25] T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: Proc. of POPL, 2002, pp. 58–70.

[26] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385–394.

[27] J. Kruskal, Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture, Trans. Am. Math. Soc. 95 (1960) 210–225.

[28] C. Lee, N. Jones, A. Ben-Amram, The size-change principle for program termination, in: Proc. of POPL'01, SIGPLAN Not. 28 (2001) 81–92.

[29] M. Leuschel, Homeomorphic embedding for online termination of symbolic methods, in: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, in: Lecture Notes on Computer Science, vol. 2566, Springer, 2002, pp. 379–403.

[30] M. Leuschel, B. Martens, D. De Schreye, Controlling generalization and polyvariance in partial deduction of normal logic programs, ACM Trans. Program. Lang. Syst. 20 (1) (1998) 208–258.
[31] J. Lloyd, Foundations of Logic Programming, second edition, Springer-Verlag, Berlin, 1987.
[32] J. Lloyd, J. Shepherdson, Partial evaluation in logic programming, J. Log. Program. 11 (1991) 217–242.
[33] Z. Manna, A. Pnueli, Temporal Verification of Reactive Systems — Safety, Springer, 1995.
[34] B. Martens, J. Gallagher, Ensuring global termination of partial deduction while allowing flexible polyvariance, in: Proc. of ICLP'95, MIT Press, 1995, pp. 597–611.
[35] N. Nishida, G. Vidal, Termination of narrowing via termination of rewriting, Appl. Algebra Eng. Commun. Comput. 21 (3) (2010) 177–225.
[36] C. Otto, M. Brockschmidt, C. von Essen, J. Giesl, Automated termination analysis of Java bytecode by term rewriting, in: C. Lynch (Ed.), Proc. of RTA 2010, in: LIPIcs, vol. 6, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2010, pp. 259–276.
[37] G. Plotkin, Building-in equational theories, Mach. Intell. 7 (1972) 73–90.
[38] A. Podelski, A. Rybalchenko, A complete method for the synthesis of linear ranking functions, in: B. Steffen, G. Levi (Eds.), Proceedings of 5th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2004, Venice, January 11–13, 2004, in: Lecture Notes on Computer Science, vol. 2937, Springer, 2004, pp. 239–251.
[39] A. Podelski, A. Rybalchenko, Transition invariants, in: Proc. of LICS'04, IEEE Computer Society, 2004, pp. 32–41.
[40] A. Podelski, A. Rybalchenko, Transition predicate abstraction and fair termination, ACM Trans. Program. Lang. Syst. 29 (3).
[41] F. Ramsey, On a problem of formal logic, in: Proc. of the London Mathematical Society, vol. 30, 1930, pp. 264–286.
[42] P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, R. Thiemann, Automated termination analysis for logic programs with cut, Theory Pract. Log. Program. 10 (4–6) (2010) 365–381.
[43] F. Spoto, F. Mesnard, É. Payet, A termination analyzer for Java bytecode based on path-length, ACM Trans. Program. Lang. Syst. 32 (3) (2010).
[44] G. Stix, Send in the terminator. A Microsoft tool looks for programs that freeze up, Sci. Am. 295 (6) (2006) 37.
[45] Terese, Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55, Cambridge University Press, 2003.
[46] G. Vidal, Termination of narrowing in left-linear constructor systems, in: J. Garrigue, M. Hermenegildo (Eds.), Proc. of the 9th International Symposium on Functional and Logic Languages (FLOPS 2008), in: Lecture Notes on Computer Science, vol. 4989, Springer, 2008, pp. 113–129.
[47] G. Vidal, Closed symbolic execution for verifying program termination, in: Proc. of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012), IEEE, 2012, pp. 34–43, available from http://users.dsic.upv.es/~gvidal.