A Generic Approach to Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

A Generic Framework for Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

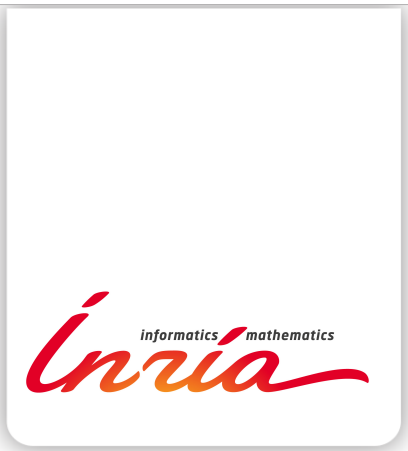
RESEARCH REPORT

N° 8189

December 2012

Project-Team Dart

ISSN 0249-6399 ISRN INRIA/RR--8189--FR+ENG



A Generic Framework for Symbolic Execution

Andrei Arusoaie*, Dorel Lucanu†, Vlad Rusu‡

Project-Team Dart

Research Report n° 8189 — December 2012 — 21 pages

Abstract: We propose a generic, language-independent symbolic execution approach for languages endowed with a formal operational semantics based on term rewriting. Starting from the definition of a language $\mathcal{L}$, a new definition $\mathcal{L}^{sym}$ is automatically generated, which has the same syntax, but whose semantics extends $\mathcal{L}$'s data domains with symbolic values and adapts the semantical rules of $\mathcal{L}$ to deal with the new domains. Then, the symbolic execution of $\mathcal{L}$ programs is the concrete execution of the corresponding $\mathcal{L}^{sym}$ programs, i.e., the application of the rewrite rules in the semantics of $\mathcal{L}^{sym}$. We prove that the symbolic execution thus defined has the adequate properties normally expected from it, and illustrate the approach on a simple imperative language defined in the $\mathbb{K}$ framework. A prototype symbolic execution engine also written in $\mathbb{K}$ is presented.

Key-words: Symbolic Execution, Term Rewriting, $\mathbb{K}$ framework.

* University of Iasi, Romania
† University of Iasi, Romania
‡ Inria Lille Nord Europe

RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Un cadre général pour l'exécution symbolique

Résumé : Nous proposons un cadre général pour l'exécution symbolique de programmes écrits dans des langages munis de sémantiques formelles définies par réécriture de termes. Partant d'une définition d'un langage $\mathcal{L}$, on construit automatiquement la définition d'un nouveau langage $\mathcal{L}^{sym}$ qui a la même syntaxe que $\mathcal{L}$ mais qui étend les types de données de $\mathcal{L}$ avec des valeurs symboliques, et dont les règles sémantiques sont adaptées pour traiter les valeurs symboliques. L'exécution symbolique des programmes de $\mathcal{L}$ est alors définie comme l'exécution habituelle des programmes de $\mathcal{L}^{sym}$, c'est à dire, l'application des règles de la sémantique de $\mathcal{L}^{sym}$ aux programmes de $\mathcal{L}$ plongés dans $\mathcal{L}^{sym}$. Nous démontrons que l'exécution symbolique possède les propriétés attendues, et illustrons l'approche sur un langage impératif simple défini dans la $\mathbb{K}$ *framework*. Nous présentons également un prototype de moteur symbolique implémenté en $\mathbb{K}$.

Mots-clés : Exécution symbolique, réécriture de termes, $\mathbb{K}$ *framework*.

1 Introduction

Symbolic execution is a well-known program analysis technique introduced in 1976 by James C. King [11]. Since then, it has proved its usefulness for testing, verifying, and debugging programs. Symbolic execution consists in providing programs with symbolic inputs, instead of concrete ones, and the execution is performed by processing expressions involving the symbolic inputs [19]. The main advantage of symbolic execution is that it allows reasoning about multiple concrete executions of a program, and its main disadvantage is the state space explosion determined by decision statements and loops. Recently, the technique has found renewed interest in the formal methods community due to new algorithmic developments and progress in decision procedures. Current applications of symbolic execution include automated test input generation [13], [26], invariant detection [18], model checking [10], or proving program partial correctness [25], [6].

The *state* of a symbolic program execution typically contains the next statement to be executed, symbolic values of program variables, and the *path condition*, which constrains past and present values of those variables (i.e., constraints on the symbolic values are accumulated on the path taken by the execution for reaching the current instruction). The states, and the transitions between them, generate a *symbolic execution tree*. When the control flow of a program is determined by symbolic values (e.g., the next instruction to be executed is a if-statement, whose Boolean condition depends on symbolic values), then there is a branching in the tree. The path condition is then used to discriminate among branches.

Two of the most important properties expected of symbolic execution are:

Coverage: for every concrete execution there is a corresponding symbolic one;

Precision: for every symbolic execution there is a corresponding concrete one;

where two executions are said to be corresponding if they take the same path.

In this paper we propose a generic, language independent symbolic execution approach that, under some reasonable conditions, has the above properties.

1.1 Related Work

There are many tools for performing symbolic execution for specific programming languages. Java PathFinder [19] is a complex symbolic execution tool which uses a model checker to explore different symbolic execution paths. The approach is applied to Java programs and it can handle recursive input data structures, arrays, preconditions, and multithreading. Java PathFinder can access through an interface several Satisfiability Modulo Theories (SMT) solvers, and the user can also choose between multiple decision procedures.

One interesting approach consists is combining concrete and symbolic execution, also known as *concolic* execution. First, some concrete values are given as input and these determine an execution path. When the program encounters a decision point, the paths not taken by concrete execution are explored symbolically. This type of analysis has been implemented by several tools for performing dynamic test generation: DART [9], CUTE [23], EXE [3], PEX [8], [4].

Symbolic execution has initially been used in automated test generation [11]. The main goal of testing is to achieve large code coverage, meaning the exploration of as many statements and code branches as possible. Symbolic execution is well suited for this since it is driven by the control flow of a program. Test sequence generation is another application of symbolic execution, consisting in generating all code sequences which explore different paths [12]. Symbolic execution is a mainly testing methodology but it can be useful for proving program correctness in case there is an upper bound for the number executions of each loop. Otherwise, loops must be annotated with invariants. There are several tools (e.g. Smallfoot [2, 27]) which use symbolic execution together with separation logic to prove Hoare triples. There are also approaches which tend

to detect automatically invariants in programs([18], [22]). Another useful symbolic execution application is the static detection of runtime errors. The main idea to perform symbolic execution on a program until a state is reached where an error occurs: null-pointer dereference, division by zero, etc.

Another body of related work is symbolic execution in term-rewriting systems. The technique called *narrowing*, initially used for solving equation systems in abstract datatypes, has been extended for solving reachability problems in term-rewriting systems and have sucessfully been applied to the analysis of security protocols [17]. Such analyses relies on powerful unification-modulo-theories algoritms [7], which works well for security protocols since there are unification algorithms modulo the theories involved there (exclusive-or, exponentiation, ...). This is not so for general programming languages, where datatypes can be arbitrary. In our approach we replace unification by a combination of matching (with possibly altered, yet equivalent) rewrite rules and calls to SMT solvers.

Our contribution. Most of the existing tools and methodologies have been developed for specific programming languages, and most of them are not based on formal semantics. In this paper we present a general, language-independent approach for symbolic execution, based on a language's formal semantics defined using term rewriting. Most existing operational semantics styles (small-step, big-step, reduction with evaluation contexts, ...) have beed shown to be representable in this way [24].

We start by identifying the main ingredients for defining programming language in an algebraic and term-rewriting setting: a signature, a model of that signature, including interpretations of data, and a set of rewrite rules. We distinguish between data, which are used by, but are not part of programming languages, and non-data (e.g., statements), which are part of a language's definition. If the data are specified equationally then our definitions are rewriting-logic specifications [16], but unlike rewriting logic, we do not assume anything about how data is defined; this allows us to focus on the language definition itself and saves us a lot of technical complications. Then, starting from the definition of a language $\mathcal{L}$, a new language definition $\mathcal{L}^{sym}$ is automatically generated, with the same syntax as $\mathcal{L}$, but whose semantics extends $\mathcal{L}$'s datatypes with symbolic values and adapts the semantical rules of $\mathcal{L}$ to handle the symbolic values.

By definition, the symbolic semantics of $\mathcal{L}$ is the semantics of $\mathcal{L}^{sym}$, and symbolic execution of programs in $\mathcal{L}$ is the (usual) execution of $\mathcal{L}^{sym}$, i.e., the application of semantical rules of $\mathcal{L}^{sym}$ to the corresponding symbolic programs.

We prove that symbolic execution has the (coverage and precision) properties presented above. We illustrate the approach on a simple imperative language IMP whose operational semantics is given in the $\mathbb{K}$ [20] semantic framework. We implement the approach in $\mathbb{K}$ and use the Z3 solver [5] for path conditions, and demonstrate the implementation on sime IMP programs.

The rest of the paper is organised as follows: Section 2 introduces of our running example (a simple imperative language IMP) and its definition in $\mathbb{K}$. Section 3 introduces our framework for language definitions. Sections 4 shows how the definition of a language $\mathcal{L}$ can be automatically extended to that of a language $\mathcal{L}^{sym}$ by extending the data of $\mathcal{L}$ with symbolic values, and the rules of $\mathcal{L}$ with means to handle those symbolic values. We also relate our matching-based approach to symbolic execution with related works based on unification. Section 5 deal with the symbolic semantics and with its relation to the concrete semantics, establishing the coverage and precision results stated in this introduction. Section 6 describes an implementation of our approach and its application to some IMP programs. Conclusions and future work plans are given in Section 7.

Id ::= domain of identifiers
Int ::= domain of integer numbers (including operations)
$Bool$ ::= domain of boolean constants (including operations)

$AExp ::= Int$
$\quad | \; Id$
$\quad | \; AExp$ `/` $AExp$ [strict]
$\quad | \; AExp$ `*` $AExp$ [strict]
$\quad | \; AExp$ `+` $AExp$ [strict]
$\quad | \; (AExp)$

$BExp ::= Bool$
$\quad | \; AExp$ `<=` $AExp$ [strict]
$\quad | \;$ `not` $BExp$ [strict]
$\quad | \; BExp$ `and` $BExp$ [strict(1)]
$\quad | \; (BExp)$

$Stmt ::=$ `skip`
$\quad | \; Id$ `:=` $AExp$
$\quad | \;$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$ [strict(1)]
$\quad | \;$ `{` $Stmt$ `}`
$\quad | \;$ `while` $BExp$ `do` $Stmt$
$\quad | \; Stmt$ `;` $Stmt$

$Code ::= Id \mid Int \mid Bool \mid AExp \mid BExp \mid Stmt \mid Code \curvearrowright Code$

Figure 1: $\mathbb{K}$ Syntax of IMP

```
if ( a <= b )
  then if (a <= c)
    then min := a
    else min := b
  else if (b <= c)
    then min := b
    else min := c;
```

Figure 2: An IMP program: computing the minimum of three numbers

$$Cfg \quad ::= \langle \langle Code \rangle_{\mathsf{k}} \langle Map_{Id,Int} \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$$

Figure 3: $\mathbb{K}$ Configuration of IMP

2 A Simple Imperative Language and its Definition in $\mathbb{K}$

Our running example is IMP, a simple imperative language intensively used in research papers. The syntax of IMP is described in Figure 1 and is mostly self-explained since it uses a BNF notation. The statements of the language are either assignments, *if* statements, *while* loops, *skip* (i.e., the empty statement), or blocks of statements. The attribute *strict* in some production rules means the arguments of the annotated expression/statement are evaluated before the expression/statement itself. If *strict* is followed by a list of natural numbers then it only concerns the arguments whose positions are present in the list.

The example shown in Figure 2 is a simple IMP program which computes the minimum of the values stored in the variables `a`, `b`, `c`. The program is intentionally incorrect when $\texttt{a} \leq \texttt{b}$ and $\texttt{a} > \texttt{c}$; in this case, `min` is set to `b` instead of `c`. We show later in the paper how the erroneous case is detected by symbolic execution.

The operational semantics of IMP is given as a set of (possibly conditional) rewrite rules. The terms to which rules apply are called *configurations*. Configurations typically contain the program to be executed, together with any additional information required for program execution. The structure of a configuration depends of the language being defined; for IMP, it consists only of the program code to be executed and an environment mapping variables to values. Configurations are written in $\mathbb{K}$ as nested structures of *cells*: for IMP, a top cell `cfg`, having a subcell `k` contating the code and a subcell `env` containing the environment (cf. Figure 3). The code inside the `k` cell is represented as a list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \ldots$ to be executed in the given order.

$$\langle\langle I_1 \texttt{ + } I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ * } I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 *_{Int} I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ / } I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1 /_{Int} I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ <= } I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 \leq_{Int} I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \mathit{true} \texttt{ and } B \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle B \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \mathit{false} \texttt{ and } B \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false} \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{not } B \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \neg B \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{skip} \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle S_1 ; S_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \curvearrowright S_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{\{ } S \texttt{ \}} \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } \mathit{true} \texttt{ then } S_1 \texttt{ else } S_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } \mathit{false} \texttt{ then } S_1 \texttt{ else } S_2 \rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2 \rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{while } B \texttt{ do } S \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow$$
$$\quad \langle\langle \texttt{if } B \texttt{ then\{ } S \texttt{ ;while } B \texttt{ do } S \texttt{ \}else skip} \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle X \ \cdots\rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I \ \cdots\rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle X \texttt{ := } I \ \cdots\rangle_{\mathsf{k}} \langle X \mapsto \_ \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \ \cdots\rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}}$$

Figure 4: $\mathbb{K}$ Semantics of IMP

Computation tasks are typically statements and expressions. The environment in the `env` cell is a multiset of bindings of variable to values, e.g., $\texttt{a} \mapsto 3$.

The semantics of IMP is shown in Figure 4. Each rewrite rule from the semantics specifies how the configuration evolves when the first computation task from the `k` cell is executed. Dots in a cell mean that the rest of the cell remains unchanged. Most syntatical constructions require one semantical rule. The exceptions are the conjunction operation and the `if` statement, which have Boolean arguments and require two rules each (one rule per Boolean value).

In addition to the rules shown in Figure 4 the semantics of IMP includes additional rules induced by the *strict* attribute. We show only the case of the `if` statement, which is strict in the first argument. The evaluation of this argument is achieved by executing the following rules:

$$\langle\langle \texttt{if } BE \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle BE \curvearrowright \texttt{if } \square \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B \curvearrowright \texttt{if } \square \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$

Here, BE ranges over $BExp \setminus \{\mathit{false}, \mathit{true}\}$, B ranges over the Booleean values $\{\mathit{false}, \mathit{true}\}$, and $\square$ is a special variable, destined to receive the value of BE once it is computed, typically, by the other rules in the semantics.

3 The Ingredients of a Language Definition

In this section we identify the ingredients of a formal language definition in an algebraic and term-rewriting setting. The concepts are then explained on the $\mathbb{K}$ definition of IMP. We assume the reader is familiar with the basics of algebraic specification, rewriting, and First-Order Logic (abbreviated FOL in this paper). A programming language $\mathcal{L}$ can be defined as a triple $(\Sigma, \mathcal{T}, \mathcal{S})$, consisting of:

1. A many-sorted algebraic signature Σ, which includes at least a sort *Cfg* for configurations

and a subsignature Σ^{Bool} for Booleans with their usual constants and operations. Σ may also include other subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all data sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{Data}$. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise infinite set of variables *Var*, let $T_\Sigma(Var)$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(Var)$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t.

2. A Σ-algebra $\mathcal{T}$. Let $\mathcal{T}_s$ denote the elements of $\mathcal{T}$ that have the sort s; the elements of $\mathcal{T}_{Cfg}$ are called *configurations*. $\mathcal{T}$ interprets the data sorts (those included in the subsignature Σ^{Data}) according to some Σ^{Data}-algebra $\mathcal{D}$[1].

   $\mathcal{T}$ interprets the non-data sorts as sets of ground terms over the signature

   $$(\Sigma \setminus \Sigma^{Data}) \cup \bigcup_{d \in Data} \mathcal{D}_d \tag{1}$$

   where $\mathcal{D}_d$ denotes the carrier set of the sort d in the algebra $\mathcal{D}$, and the elements of $\mathcal{D}_d$ are added to the signature $\Sigma \setminus \Sigma^{Data}$ as constants of sort d[2].

   Any *valuation* $\rho : Var \to \mathcal{T}$ is extended to a (homonymous) Σ-algebra morphism $\rho : T_\Sigma(Var) \to \mathcal{T}$. The interpretation of a ground term t in $\mathcal{T}$ is denoted by $\mathcal{T}_t$. If $b \in T_{\Sigma,Bool}(Var)$ then we write $\rho \models b$ iff $\rho(b) = \mathcal{D}_{true}$. For simplicity, we often write in the sequel *true*, *false* instead of $\mathcal{D}_{true}, \mathcal{D}_{false}$.

3. A set $\mathcal{S}$ of rewrite rules, whose definition is given later in the section.

We explain these concepts on the IMP example. Nonterminals from the syntax (*Int*, *Bool*, *AExp*, ...) are sorts in Σ. Each production from the syntax defines an operation in Σ; for instance, the production $AExp ::= AExp + AExp$ defines the operation $\_+\_ : AExp \times AExp \to AExp$. These operations define the constructors of the result sort. For the configuration sort *Cfg*, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : Code \times Map_{Id,Int} \to Cfg$. The expression $\langle\langle X := I \curvearrowright C\rangle_{\mathsf{k}}\langle X \mapsto 0\ Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a term of $T_{Cfg}(Var)$, where X is a variable of sort *Id*, I is a variable of sort *Int*, C is a variable of sort *Code* (the rest of the computation), and *Env* is a variable of sort $Map_{Id,Int}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets *Int* as the set of integers, the operations like $+_{Int}$ (cf. Figure 4) as the corresponding usual operation on integers, *Bool* as the set of Boolean values $\{false, true\}$, the operation like $\wedge$ as the usual Boolean operations, the sort $Map_{Id,Int}$ as the multiset of maps $X \mapsto I$, where X ranges over identifiers *Id* and I over the integers. The other sorts, *AExp*, *BExp*, *Stmt*, and *Code*, are interpreted in the algebra $\mathcal{T}$ as ground terms over a modification of the form (1) of the signature Σ, in which data subterms are replaced by their interpretations in $\mathcal{D}$. For instance, the term `if` $1 >_{Int} 0$ `then skip else skip` is intepreted as `if` *true* `then skip else skip` provided $\mathcal{D}_{1>_{Int}0} = \mathcal{D}_{true} (= true)$.

We now formally introduce the notions required for defining semantical rules.

[1] A possible definition for $\mathcal{D}$ is the following one. Assume the data are defined equationally, i.e., there is a finite set of equations E^{Data} defining the data sorts and the operations on them. Then, $\mathcal{D}$ can be defined as the initial algebra of the equational specification $(\Sigma^{Data}, E^{Data})$. We chose not to impose that data be defined in any particular way, since they are not part of the programming language's definition.

[2] If data were defined equationally - cf. previous footnote - then $\mathcal{T}$ would be defined as the initial algebra of the equational specification (Σ, E^{Data}). Again, we did not choose this approach in order to avoid assumptions about how the data are defined.

Definition 1 (pattern [21]) *A* pattern *is an expression of the form* $\pi \wedge b$*, where* $\pi \in T_{\Sigma,Cfg}(Var)$ *are* basic patterns*,* $b \in T_{\Sigma,Bool}(Var)$*, and* $var(b) \subseteq var(\pi)$*. If* $\gamma \in T_{Cfg}$ *and* $\rho : Var \to \mathcal{T}$ *we write* $(\gamma, \rho) \models \pi \wedge b$ *for* $\gamma = \rho(\pi)$ *and* $\rho \models b$*.*

A basic pattern π defines a set of (concrete) configurations, and the condition b gives additional constraints these configurations must satisfy. In [21] patterns are encoded as FOL formulas, hence the conjunction notation $\pi \wedge b$. In this paper we keep the notation but separate basic patterns from constraining formulas. We identify basic patterns π with paterns $\pi \wedge \mathit{true}$.

Sample patterns are $\langle\langle I_1 + I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \,/\, I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0$.

Definition 2 (semantical rule and transition system) *A* rule *is a pair of patterns of the form* $l \wedge b \Rightarrow r$ *(note that* r *is in fact the pattern* $r \wedge \mathit{true}$*). Any set* $\mathcal{S}$ *of rules defines a labelled transition system* $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ *such that* $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$ *iff* $\alpha \triangleq (l \wedge b \Rightarrow r) \in \mathcal{S}$ *and* $\rho : Var \to \mathcal{T}$ *are such that* $(\gamma, \rho) \models l \wedge b$ *and* $(\gamma', \rho) \models r$*.*

We note that if data were defined as initial models of certain equational specifications then the transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ could be defined as the initial model of the rewriting-logic specification obtained by adding the rules $\mathcal{S}$ to those equational specifications. Not defining the data in this way gives us some freedom and saves us some technical difficulties, as will be shown in the next section.

4 Symbolic Semantics by Data Extension

We show in this section how, given a definition $(\Sigma, \mathcal{T}, S)$ of a language $\mathcal{L}$ a new definition $(\Sigma^{sym}, \mathcal{T}^{sym}, \mathcal{S}^{sym})$ for a language $\mathcal{L}^{sym}$ is automatically generated. The new language $\mathcal{L}^{sym}$ has the same syntax, and its semantics extends $\mathcal{L}$'s data domains with symbolic values and adapts the semantical rules of $\mathcal{L}$ to deal with the new domains. Then, the symbolic execution of $\mathcal{L}$ programs is the concrete execution of the corresponding $\mathcal{L}^{sym}$ programs, i.e., the application of the rewrite rules in the semantics of $\mathcal{L}^{sym}$. Building the definition of $\mathcal{L}^{sym}$ amounts to:

1. extending the signature Σ to a symbolic signature Σ^{sym};
2. extending the Σ-algebra $\mathcal{T}$ to a Σ^{sym}-algebra $\mathcal{T}^{sym}$;
3. turning the concrete rules $\mathcal{S}$ into symbolic rules $\mathcal{S}^{sym}$.

We then obtain the symbolic transition system $(\mathcal{T}_{Cfg^{sym}}^{sym}, \Rightarrow_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}})$ by using Definitions 1,2 for $\mathcal{L}^{sym}$, just like the transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ was defined for $\mathcal{L}$. Section 5 then deals with the relations between the two transition systems.

4.1 Extending the Signature Σ to a Symbolic Signature Σ^{sym}

We fix a sort-wise set of *symbolic values SymVal*, which are fresh with respect to Σ, and let $\Sigma(SymVal)$ denote the signature Σ enriched with *SymVal* as constant declarations of the corresponding sorts. We call any (data) sort s for which there is a symbolic value of sort s a *symbolically extensible sort.*

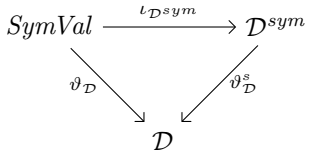


Figure 5: Diagram Characterising $\vartheta^s_{\mathcal{D}} : \mathcal{D}^{sym} \to \mathcal{D}$

Assumption The symbolically extensible sorts are among the data sorts. Non-data symbolically extensible sorts are left for study in future work.

The symbolic signature Σ^{sym} includes the signature $\Sigma(SymVal)$ enriched with a new sort *Fol* for FOL formulas, together with all the required operations that allow FOL formulas to be written as ground terms of sort *Fol*.
Example For the IMP example this extension amounts to declaring

$$Fol ::= Bool \mid (\forall\ SymVal)\ Fol \mid (\exists\ SymVal)\ Fol \mid Fol \wedge Fol \mid \neg\, Fol$$

□

Then, we enrich the signature with a new operation symbol `unsat` : $Fol \to Bool$, whose intented meaning is to identify a set of unsatisfiable formulas. Next, we extend Σ^{sym} with a sort Cfg^{sym} and a constructor $\langle \_, \_ \rangle : Cfg \times Fol \to Cfg^{sym}$, for building symbolic configurations that are pairs consisting of configurations over symbolic data and a FOL formula denoting path conditions. Finally, we extend the set of variables *Var* with infinitely many variables of sort *Fol*.
Example For the IMP example we enrich the configuration with a new cell:

$$Cfg^{sym} ::= \langle \langle Code \rangle_{\mathsf{k}} \langle Map_{Id,Int} \rangle_{\mathsf{env}} \langle Fol \rangle_{\mathsf{cnd}} \rangle_{\mathsf{cfg}}$$

where the new cell `cnd` includes a formula meant to express a path condition. □

4.2 Extending the Model $\mathcal{T}$ to a Symbolic Model $\mathcal{T}^{sym}$

We first deal with the *symbolic domain* $\mathcal{D}^{sym}$, a Σ^{Data}-algebra with the following properties: 1) the Σ^{Data}-algebra $\mathcal{D}$ is a sub-algebra of $\mathcal{D}^{sym}$, 2) there is an injection $\iota_{\mathcal{D}^{sym}} : SymVal \to \mathcal{D}^{sym}$, and 3) for any valuation $\vartheta_{\mathcal{D}} : SymVal \to \mathcal{D}$ there is a unique algebra morphism $\vartheta^s_{\mathcal{D}} : \mathcal{D}^{sym} \to \mathcal{D}$, such that the diagram in Figure 5 commutes. The diagram says that symbolic values are data in $\mathcal{D}^{sym}$ via the injection $\iota_{\mathcal{D}^{sym}}$, and that any interpretation $\vartheta_{\mathcal{D}}$ of the symbolic values as concrete data is uniquely extended to an algebra morphism $\vartheta^s_{\mathcal{D}}$ that assigns concrete values to symbolic values. For instance, $\mathcal{D}^{sym}$ can be the Σ^{Data}-algebra of expressions built over $\mathcal{D}$, where *SymVal* play the role of variables, or the quotient of this algebra modulo the congruence defined by some set of equations E^{sym} (which can be used in practice as simplification rules for symbolic expressions).

We leave some freedom in choosing the symbolic domain, to allow the use of decision procedures or other efficient means for handling symbolic artifacts.

We now give the interpretation for the remaining syntax that Σ^{sym} introduced with respect to Σ. The interpretation of ground terms of sort *Fol* are the corresponding FOL formulas. The operation symbol `unsat` is interpreted as a homonymous predicate and is assumed to be *sound*: for any FOL formula ϕ, if $\texttt{unsat}(\phi) = true$ then ϕ is unsatisfiable. The converse property: if ϕ is unsatisfiable then $\texttt{unsat}(\phi) = true$, is called *completeness*[3] and is only required for the

[3]Note that soundness and completeness are relative to the (usual) semantics of FOL.

(essentially, theoretical) precision result regarding symbolic execution[4]. The Σ-terms of sort *Cfg* are interpreted in $\mathcal{T}^{sym}$ like in $\mathcal{T}$, and the terms of sort Cfg^{sym} are interpreted as pairs $\langle \gamma^s, \phi \rangle$, where $\langle \_, \_ \rangle : \mathcal{T}^{sym}_{Cfg} \times \mathcal{T}^{sym}_{Fol} \to \mathcal{T}^{sym}_{Cfg^{sym}}$ is the interpretation of the operation symbol $\langle \_, \_ \rangle : Cfg \times Fol \to Cfg^{sym}$.

Definition 3 (Symbolic Configuration and Satisfaction Relation) *A concrete configuration $\gamma \in \mathcal{T}_{Cfg}$* satisfies *a symbolic configuration $\langle \gamma^s, \phi \rangle \in \mathcal{T}^{sym}_{Cfg^{sym}}$, written $\gamma \models \langle \gamma^s, \phi \rangle$, if there is $\vartheta : SymVal \to \mathcal{T}$ such that $\gamma = \vartheta(\gamma^s)$ and $\vartheta \models \phi$.*

In the above definition, the notation $\vartheta \models \phi$ means that the valuation θ of *SymVal* satisfies the FOL formula ϕ according to the usual satisfaction relation of FOL. *Example* The concrete configuration

$\gamma \triangleq \langle \langle \texttt{if } true \texttt{ then skip else skip} \rangle_{\mathsf{k}} \langle \emptyset \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$

satisfies the symbolic configuration

$\langle \gamma^s, \phi \rangle \triangleq \langle \langle \texttt{if } b^s \texttt{ then skip else skip} \rangle_{\mathsf{k}} \langle \emptyset \rangle_{\mathsf{env}} \langle b^s = true \rangle_{\mathsf{cnd}} \rangle_{\mathsf{cfg}}$

where the valuation ϑ is $b^s \mapsto true$, and in this case it is directly computed from the formula $\phi \triangleq (b^s = true)$ included in the additional `cnd` cell. Here b^s is a symbolic value of sort *Bool*. □

4.3 Turning the Concrete Rules $\mathcal{S}$ into Symbolic Rules $\mathcal{S}^{sym}$

We show how to automatically build the symbolic-semantics rules $\mathcal{S}^{sym}$ from the concrete semantics-rules $\mathcal{S}$, by applying the three steps described below.

We first make the following assumption: the left-hand sides of rules do not contain operations on symbolically extensible sorts. For example, if the data sort *Map* is symbolically extensible, then the last two rules from the IMP semantics (cf. Figure 4):

$$\langle \langle X \ \cdots \rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}} \Rightarrow \langle \langle I \ \cdots \rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}}$$
$$\langle \langle X \ \texttt{:=} \ I \ \cdots \rangle_{\mathsf{k}} \langle X \mapsto \_ \ \cdots \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}} \Rightarrow \langle \langle \ \cdots \rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}}$$

violate the assumption, because $\langle X \mapsto I \ \cdots \rangle_{\mathsf{env}}$ is a shortcut for $\langle (X \mapsto I) M \rangle_{\mathsf{env}}$ for some map variable M, where the juxtaposition operation is map composition.

The assumption can sometimes be made to hold by transforming the rules into equivalent ones; for example, the two problematic rules can be rewritten

$$\langle \langle X \ \cdots \rangle_{\mathsf{k}} \langle M \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}} \wedge (lookup(M, X) = I) \Rightarrow \langle \langle I \ \cdots \rangle_{\mathsf{k}} \langle M \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}}$$
$$\langle \langle X \ \texttt{:=} \ I \ \cdots \rangle_{\mathsf{k}} \langle M \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}} \Rightarrow \langle \langle \ \cdots \rangle_{\mathsf{k}} \langle update(M, X, I) \rangle_{\mathsf{env}} \ \cdots \rangle_{\mathsf{cfg}}$$

where $\_ = \_$ is a function symbol that returns a Boolean, assumed to exist for all data sorts, and interpreted as returning *true* iff its arguments are equal, and *lookup*(), *update*() are the usual lookup and updating functions for maps, which have then to be defined in a subsignature Σ^{Map} of Σ^{Data}. If *Map* is not symbolically extensible then the original semantics satisfies our assumption.

[4] This is is where defining data as initial algebras of equational specifications would be too restrictive: since `unsat` returns a Boolean it needs to be fully defined, as required by initial algebras. Thus, a sound and complete interpretation of `unsat` in $\mathcal{T}^{sym}$ is impossible if $\mathcal{T}^{sym}$ was an initial algebra, since satisfiability in FOL is undecidable. We did not impose the definition of data in this way, and are free to interpret `unsat` as a decision oracle for FOL when needed later in the paper for theoretical reasons.

4.3.1 1. Linearising Rules

A rule is (left) linear if any variable occurs at most once in its left-hand side. A nonlinear rule can always be turned into an equivalent linear one, by renaming some variables and adding equalities between the renamed variables and the original ones to the rule's condition. Recall the last two rules from the original IMP semantics (Fig. 4). These rules are non-linear because variable X appears twice in the left-hand side, once in the $\langle ... \rangle_{\mathsf{k}}$ cell and once in the $\langle ... \rangle_{\mathsf{env}}$ cell. To linearise them we just add a new variable, say X', and a condition, $X = X'$:

$$\langle\langle X \ \cdots\rangle_{\mathsf{k}} \langle X' \mapsto I \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}} \wedge X = X' \Rightarrow \langle\langle I \ \cdots\rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle X := I \ \cdots\rangle_{\mathsf{k}} \langle X' \mapsto \_ \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}} \wedge X = X' \Rightarrow \langle\langle \ \cdots\rangle_{\mathsf{k}} \langle X \mapsto I \ \cdots\rangle_{\mathsf{env}} \ \cdots\rangle_{\mathsf{cfg}}$$

This process is entirely automatic. Of course, there are other ways of linearising rules, e.g., the rules transformed using *lookup*() and *update*() also shown above.

4.3.2 2. Replacing Constants by Variables

Let $Cpos(l)$ be the set of positions ω[5] of the term l such that l_ω is a constant of a symbolically extensible sort. The next step of our rule transformation consists in replacing all the constants of symbolically extensible sorts by fresh variables. The purpose of this step is to make rules match any configuration, including the symbolic ones.

Thus, we transform each rule $l \wedge b \Rightarrow r$ into the rule

$$[l_\omega / X_\omega]_{\omega \in Cpos(l)} \wedge b \wedge \bigwedge_{\omega \in Cpos(l)} (X_\omega = l_\omega) \Rightarrow r,$$

where each X_ω is a new variable of same sort as l_ω

Example Consider the following rule for *if* from the IMP semantics:

$$\langle\langle \texttt{if } \mathit{true} \texttt{ then } S_1 \texttt{ else } S_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$

Assuming Boolean is a symbolically extensible sort, we replace the constant *true* with a Boolean variable B, and add the condition $B = \mathit{true}$:

$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \wedge B = \mathit{true} \Rightarrow \langle\langle S_1 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$

□

4.3.3 3. Adding Formulas to Configurations and Rules

Let `unsat` be the unsatisfiability predicate in Σ^{sym}. The last transformation step consists in transforming each rule $l \wedge b \Rightarrow r$ in $\mathcal{S}$ obtained after the previous steps, into the following one:

$$\langle l, \psi \rangle \wedge \neg \, \texttt{unsat}(\psi \wedge b) \Rightarrow \langle r, \psi \wedge b \rangle \tag{2}$$

where $\psi \in \mathit{Var}$ is a variable of sort *Fol* and $\langle \_, \_ \rangle$ is the operation in Σ^{sym}. Intuitively, this means that a symbolic transition if performed on a symbolic configuration is the conjunction of the symbolic configuration's path condition and the concrete rule's condition is not unsatisfiable. Indeed, this is what happens in the transition system since we chose to interpret `unsat` as a sound predicate. *Example* The last rule for `if` from the (already transformed) IMP semantics is further transformed into the following rule in $\mathcal{S}^{sym}$:

$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \langle \psi \rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}} \wedge \neg \, \texttt{unsat}(\psi \wedge B) \Rightarrow \langle\langle S_1 \cdots\rangle_{\mathsf{k}} \langle \psi \wedge B \rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}}$$

□

[5] For the notion of position in a term and other rewriting-related notions, see, e.g., [1].

4.4 Defining the Symbolic Transition System $(\mathcal{T}^{sym}_{Cfg^{sym}}, \Rightarrow^{\mathcal{T}^{sym}}_{\mathcal{S}^{sym}})$

The triple $(\Sigma^{sym}, \mathcal{T}^{sym}, \mathcal{S}^{sym})$ defines a language $\mathcal{L}^{sym}$. Then, the transition system $(\mathcal{T}^{sym}_{Cfg^{sym}}, \Rightarrow^{\mathcal{T}^{sym}}_{\mathcal{S}^{sym}})$ can be defined using Definitions 1 and 2 applied to $\mathcal{L}^{sym}$.

For this, we note that the left-hand sides of the rules of $\mathcal{L}^{sym}$, of the form $\langle l, \psi\rangle$, are terms in $T_{\Sigma^{sym}, Cfg^{sym}}(Var)$, and that the conditions of the rules of $\mathcal{L}^{sym}$, of the form $\neg\,\mathtt{unsat}(\psi \wedge b)$, are terms in $T_{\Sigma^{sym}, Bool}(Var)$ - remember that Var has been extended to include variables of sort *Fol*. Since $l \wedge b$ is a pattern of $\mathcal{L}$, using Definition 1 applied to $\mathcal{L}$ we get $var(b) \subseteq var(l)$, and then $var(\psi \wedge b) = var(b) \cup \{\psi\} \subseteq var(l) \cup \{\psi\} = var(\langle l, \psi\rangle)$. It follows, according to Definition 1 applied to $\mathcal{L}^{sym}$, that the expression $\langle l, \psi\rangle \wedge \neg\,\mathtt{unsat}(\psi \wedge b)$ is a pattern of $\mathcal{L}^{sym}$, and then Definition 2 for $\mathcal{L}^{sym}$ gives us the transition system $(\mathcal{T}^{sym}_{Cfg^{sym}}, \Rightarrow^{\mathcal{T}^{sym}}_{\mathcal{S}^{sym}})$.

5 Relating the Concrete and Symbolic Semantics of $\mathcal{L}$

We now relate the concrete and symbolic semantics of $\mathcal{L}$, i.e., the transition systems $(\mathcal{T}_{Cfg}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ and $(\mathcal{T}^{sym}_{Cfg^{sym}}, \Rightarrow^{\mathcal{T}^{sym}}_{\mathcal{S}^{sym}})$. We prove certain simulation relations between them and obtain the coverage and precision properties as corollaries.

The following technical lemma is essential for obtaining a match betwen the left-land side l of a rule and a symbolic configuration, provided there is a match between l and a concrete configuration that satisfies that symbolic configuration.

We denote by $f|_{A'}$ the restriction of a function $f : A \to B$ to $A' \subseteq A$.

Lemma 1 *Let $l \in T_{\Sigma, Cfg}(Var)$ be the left-hand side of a rule in $\mathcal{S}$, $\rho : Var \to \mathcal{T}$ a valuation, and $\gamma^s \in \mathcal{T}^{sym}_{Cfg}$. If $\vartheta : SymVal \to \mathcal{T}$ satisfies $\vartheta(\gamma^s) = \rho(l)$ then there is a valuation $\rho^s : Var \to \mathcal{T}^{sym}$ such that $\gamma^s = \rho^s(l)$ and $\rho = \vartheta \circ \rho^s$.*

Proof Note first that by notation abuse we are using the valuation $\vartheta : SymVal \to \mathcal{T}$ in $\vartheta(\gamma^s) = \rho(l)$ and $\rho = \vartheta \circ \rho^s$, where we should be using ϑ^s, the homomorphical extension of ϑ to terms in $\mathcal{T}^{sym}$. (Remember that, by definition, $\mathcal{T}^{sym}$ consists of ground terms over a signature of the form (1) where $\mathcal{D}^{sym}$ replaces $\mathcal{D}$).

We also make the following remark: (♠) for all $n, n' \in \mathbb{N}$, for all operation symbols f, f' such that the result sort of f is not a data sort, and all elements $\tau_1, \ldots, \tau_n, \tau'_1, \ldots, \tau'_{n'} \in \mathcal{T}$, if $\mathcal{T}_f(\tau_1, \ldots, \tau_n) = \mathcal{T}_{f'}(\tau'_1, \ldots, \tau'_{n'})$ then $f = f'$, $n = n'$, and $\tau_i = \tau'_i$ for all $i = 1, \ldots, n$. This is because $\mathcal{T}_f(\tau_1, \ldots, \tau_n)$ is the interpretation of some ground term of a non-data sort, and such terms are interpreted as ground terms over a certain signature (of the form (1)). Hence, the only way such a term can be equal to some other term is by being syntactically equal to it. We also assume without loss of generality that for each variable $y \in Var \setminus var(l)$ there is a symbolic value y^s that does not occur in γ'^s, such that $\rho(y) = \vartheta(y^s)$.

We prove the lemma by establishing a more general result, where l can be any subterm of a left-hand side of a rule in $\mathcal{S}$. The proof goes by structural induction. There are three cases:

1. l is a variable x. Then we take ρ^s such that $\rho^s(x) = \gamma^s$, and $\rho^s(y) = y^s$ for $y \neq x$. First, $\rho^s(x) = \gamma^s$ is just $\rho^s(l) = \gamma^s$, which proves the first conclusion of the lemma in this case. Moreover $\rho(x) = \rho(l) = \vartheta(\gamma^s) = \vartheta(\rho^s(x))$, and for all $y \in Var \setminus \{x\}$, $\rho(y) = \vartheta(y^s) = \vartheta(\rho^s(y))$, which proves the second conclusion.

2. l is a constant c. Since the sort of c is not symbolically extensible, $\rho(c) = \mathcal{T}_c = \mathcal{T}^{sym}_c = \vartheta(\gamma^s)$. Let $\gamma'^s = f'(\gamma'^s_1, \ldots, \gamma'^s_{n'})$, thus, $\vartheta(\gamma'^s) = \mathcal{T}_{f'}(\vartheta(\gamma'^s_1), \ldots, \vartheta(\gamma'^s_{n'}))$. Using the remark (♠), this means $f' = c$ and $n' = 0$, and then $\vartheta(\gamma'^s) = \gamma'^s = \mathcal{T}_c$. We take ρ^s such that $\rho^s(y) = y^s$ for any y. Then, $\rho^s(l) = \rho^s(c) = \mathcal{T}_c = \gamma'^s$, and for all $y \in Var$, $\rho(y) = \vartheta(y^s) = (\vartheta \circ \rho^s)(y)$, which proves this case as well.

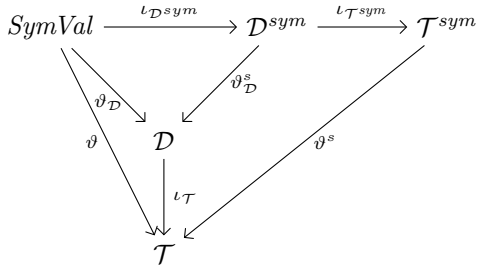


Figure 6: Diagram Characterising $\vartheta^s : \mathcal{T}^{sym} \to \mathcal{T}$. Contains Fig. 5 as subdiagram.

3. $l = f(t_1, \ldots, t_n)$. Let $\gamma'^s = f'(\gamma'^s_1, \ldots, \gamma'^s_{n'})$. Thus, we have $\rho(t) = \mathcal{T}_f(\rho(t_1), \ldots, \rho(t_n))$ $= \mathcal{T}_{f'}(\vartheta(\gamma'^s_1), \ldots, \vartheta(\gamma'^s_{n'})) = \vartheta(\gamma'^s)$ that implies $f = f'$, $n = n'$, and $\rho(t_1) = \vartheta(\gamma'^s_1)$, $\ldots$, $\rho(t_n) = \vartheta(\gamma'^s_n)$ by the remark (♠). There are $\rho^s_i : \mathit{Var} \to \mathcal{T}^{sym}$ with $\rho^s_i(t_i) = \gamma'^s_i$ and $\rho_i = \vartheta \circ \rho^s_i$ by the inductive hypothesis. Since t is linear, it follows that $\mathit{var}(t_1), \ldots, \mathit{var}(t_n)$ are pairwise disjoint and we can build a valuation $\rho^s : \mathit{Var} \to \mathcal{T}^{sym}$ with the following properties: (1) $\rho^s|_{\mathit{var}(t_i)} = \rho^s_i|_{\mathit{var}(t_i)}$ for each $i = 1, \ldots$, and (2) ρ^s equals some arbitrarily chosen ρ^s_i in the rest. We have $\rho^s(t) = f(\rho^s(t_1), \ldots, \rho^s(t_n)) = f(\rho^s_1(t_1), \ldots, \rho^s_n(t_n)) = f(\gamma'^s_1, \ldots, \gamma'^s_n) = \gamma'^s$. The equality $\rho = \vartheta \circ \rho^s$ follows by the construction of ρ^s and the inductive hypotheses.

□

Corollary 1 *$\rho^s|_{\mathit{var}(l)}$, for ρ^s given by Lemma 1, is unique for given γ^s and l.*

Proof By structural induction over l and using the observation (♠) from above. □

Lemma 1 gives us the formal ground for relating our symbolic execution with symbolic execution via unification as done in related works, e.g., [7]. The main idea is that the substitution ρ^s from Lemma 1, which depends essentially only on γ^s and l, generates a *symbolic unifier* that *subsumes* all their *concrete unifiers*.

We first note that, as stated at the beginning of the proof of Lemma 1, the actual last statement of Lemma 1 is $\rho = \vartheta^s \circ \rho^s$, where ϑ^s is the unique morphism from $\mathcal{T}^{sym}$ to $\mathcal{T}$ making the outermost diagram in Figure 6 commute. Note that this diagram contains as a subdiagram the one in Figure 5 (top left corner).

Next, we need to adapt some definitions regarding unification. For $\gamma^s \in \mathcal{T}^{sym}_{Cfg}$ and $l \in T_{\Sigma, Cfg}(\mathit{Var})$, we say that $\vartheta \uplus \rho : \mathit{SymVal} \uplus \mathit{var}(l) \to \mathcal{T}$ is a *concrete unifier* of γ^s and l if $\vartheta(\gamma^s) =_{\mathcal{T}} \rho(l)$, and that $\vartheta^s \uplus \rho^s : \mathit{SymVal} \uplus \mathit{var}(l) \to \mathcal{T}^{sym}$ is a *symbolic unifier* of γ^s and l if $\vartheta^s(\gamma^s) =_{\mathcal{T}^{sym}} \rho^s(l)$. Note that the former equality holds in $\mathcal{T}$, while the latter holds in $\mathcal{T}^{sym}$, as emphasised by the subscripts. Note also that (for simplicity) the domains of ρ and ρ^s were chosen to be $\mathit{var}(l)$ since the values to which they evaluate the variables outside $\mathit{var}(l)$ do not matter.

We say that a symbolic unifier $\vartheta^s \uplus \rho^s$ *subsumes* a concrete unifier $\vartheta \uplus \rho$ if there is $\eta : \mathcal{T}^{sym} \to \mathcal{T}$ such that $\theta = \eta \circ \theta^s$ and $\rho = \eta \circ \rho^s$. This is illustrated by the diagram in Figure 7. What Lemma 1 tells us is that the symbolic unifier $(\iota_{\mathcal{T}^{sym}} \circ \iota_{\mathcal{D}^{sym}}) \uplus \rho^s : \mathit{SymVal} \uplus \mathit{var}(l) \to \mathcal{T}^{sym}$, where $\iota_{\mathcal{T}^{sym}}$ and $\iota_{\mathcal{D}^{sym}}$ are the injections shown in the diagram in Figure 6, subsumes the concrete unifier $\vartheta \uplus \rho$ from which ρ^s can be obtained by using Lemma 1.

Indeed, we have $\vartheta = \vartheta^s \circ (\iota_{\mathcal{T}^{sym}} \circ \iota_{\mathcal{D}^{sym}})$, which is given by the diagram in Figure 6, and $\rho = \vartheta^s \circ \rho^s$, which is implied by Lemma 1, thus, $(\iota_{\mathcal{T}^{sym}} \circ \iota_{\mathcal{D}^{sym}}) \uplus \rho^s$ subsumes $\vartheta \uplus \rho$. But by Corollary 1, $\rho^s = \rho^s|_{\mathit{var}(l)}$ is unique for given l and γ^s, thus, the above reasoning could be made

for any concrete unifier of l and γ^s, and still obtain the same symbolic unifier $(\iota_{\mathcal{T}^{sym}} \circ \iota_{\mathcal{D}^{sym}}) \uplus \rho^s$ to subsume it. Hence, this symbolic unifier of γ^s and l subsumes all concrete unifiers of γ^s and l.

Thus, we do not need to compute those concrete unifiers, or most general unifiers that subsume all of them as [7] do, which is an advantage for us, since there are few theories with adequate (i.e., finitary and complete) unification algorithms. On the other hand, we rely on SMT solvers for checking parts of the rule's conditions that unification algoritms check directly - namely, those parts introduced by transformations of $\mathcal{S}$ into $\mathcal{S}^{sym}$ described earlier in this section. The bottom line is: we postpone inherent incompleteness issues to SMT solving.

The next lemma shows that the symbolic transition system forward-simulates the concrete transition system. The notion of forward simulation (and of backwards simulation, used later) is borrowed from Lynch and Vandraager [15]. We denote by $\alpha^{sym} \in \mathcal{S}^{sym}$ the rule obtained by transforming $\alpha \in \mathcal{S}$ (Section 4.3).

Lemma 2 ($(\mathcal{T}^{sym}_{Cfg^{sym}}, \Rightarrow^{\mathcal{T}^{sym}}_{\mathcal{S}^{sym}})$ forward simulates $(\mathcal{T}_{Cfg}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$) *For all configurations γ, symbolic configurations $\langle \gamma^s, \phi \rangle$ and rules $\alpha \in \mathcal{S}$, if $\gamma \models \langle \gamma^s, \phi \rangle$ and $\gamma \overset{\alpha}{\Longrightarrow}^{\mathcal{T}}_{\mathcal{S}} \gamma'$ then $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}^{\mathcal{T}^{sym}}_{\mathcal{S}^{sym}} \langle \gamma'^s, \phi' \rangle$ and $\gamma' \models \langle \gamma'^s, \phi' \rangle$, for some $\langle \gamma'^s, \phi' \rangle$.*

Proof Let $\alpha \triangleq l \wedge b \Rightarrow r$. The transition $\gamma \overset{\alpha}{\Longrightarrow}^{\mathcal{T}}_{\mathcal{S}} \gamma'$ implies there is $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models l \wedge b$ and $(\gamma', \rho) \models r$. The satisfaction $\gamma \models \langle \gamma^s, \phi \rangle$ implies there is $\vartheta : SymVal \to \mathcal{T}$ such that $\gamma = \vartheta(\gamma^s)$ and $\vartheta \models \phi$. By Lemma 1 there is $\rho^s : Var \to \mathcal{T}^{sym}$ such that $\gamma^s = \rho^s(l)$ and $\rho = \vartheta \circ \rho^s$. We have two cases:

1. $\texttt{unsat}(\phi \wedge \rho^s(b)) = \mathit{false}$. Then, the rule $\alpha^{sym} \triangleq (\langle l, \psi \rangle \wedge \neg\, \texttt{unsat}(\psi \wedge b) \Rightarrow \langle r, \psi \wedge b \rangle)$ can be applied to the symbolic configuration $\langle \gamma^s, \phi \rangle$, yielding the transition $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}^{\mathcal{T}^{sym}}_{\mathcal{S}^{sym}} \langle \rho^s(r), \phi \wedge \rho^s(b) \rangle$. Let $\gamma'^s \triangleq \rho^s(r)$, $\phi' \triangleq \phi \wedge \rho^s(b)$.

   We prove $\gamma' \models \langle \gamma'^s, \phi' \rangle$. From the hypothesis, $\gamma' = \rho(r)$, but $\rho(r) = (\vartheta \circ \rho^s)(r) = \vartheta(\rho^s(r)) = \vartheta(\gamma'^s)$ which means $\gamma' = \vartheta(\gamma'^s)$. On the other hand, $\rho \models b$ means $\rho(b) = \mathit{true}$, which implies $(\vartheta \circ \rho^s)(b) = \mathit{true}$. Thus, $\vartheta(\rho^s(b)) = \mathit{true}$, i.e., $\vartheta \models \rho^s(b)$. Using $\vartheta \models \phi$, we get $\vartheta \models \phi \wedge \rho^s(b)$, i.e., $\vartheta \models \phi'$. By Definition 3 this means $\gamma' \models \langle \gamma'^s, \phi' \rangle$. This proves the lemma in this case.

2. $\texttt{unsat}(\phi \wedge \rho^s(b)) = \mathit{true}$. This is impossible, since the soundness of `unsat` implies $\phi \wedge \rho^s(b)$ is unsatisfiable, in contradiction with $\vartheta \models \phi \wedge \rho^s(b)$ that we deduce as above from the hypotheses of the lemma. This concludes the proof.

□

Remark 1 *Note that only the soundness of* `unsat` *was used in this proof. This is important for implementation purposes, since satisfaction in FOL is undecidable, thus, sound* and *complete unsatisfiability predicates cannot be implemented; whereas sound ones are implemented in SMT provers such as Z3 that we use.*

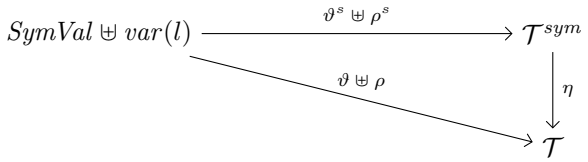


Figure 7: Subsumption Relation.

For $\beta \triangleq \beta_1 \cdots \beta_n \in \mathcal{S}^*$ we write $\gamma_0 \overset{\beta}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma_n$ for $\gamma_i \overset{\beta_{i+1}}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma_{i+1}$ for all $i = 0, \ldots, n-1$, and use a similar notation for sequences of transitions in the symbolic transition system, where we denote β^{sym} the sequence $\beta_1^{sym} \cdots \beta_n^{sym} \in \mathcal{S}^{sym\,*}$.

We can now state the coverage theorem as a corollary to the above lemma:

Corollary 2 (Coverage) *With the above notations, if* $\gamma \overset{\beta}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$ *and* $\gamma \models \langle \gamma^s, \phi \rangle$ *then there exists* $\langle \gamma'^s, \phi' \rangle$ *such that* $\gamma' \models \langle \gamma'^s, \phi' \rangle$ *and* $\langle \gamma^s, \phi \rangle \overset{\beta^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$.

Proof By induction on the length of β, using Lemma 2 for the induction step. □

The coverage corollary says that if a sequence β of rewrite rules can be executed starting in some initial configuration, the corresponding sequence of symbolic rules can be fired as well. That is, if a program can execute a certain control-flow path concretely, then it can also execute that path symbolically.

We would like, naturally, to prove the converse result (precision) based on a simulation result similar to Lemma 2: *for all configurations* γ *and symbolic configuration* $\langle \gamma^s, \phi \rangle$, *if* $\gamma \models \langle \gamma^s, \phi \rangle$ *and* $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ *then there is a configuration* γ' *such that* $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$ *and* $\gamma' \models \langle \gamma'^s, \phi' \rangle$. But this is false.

Example Consider the following (symbolic) configurations and (symbolic) rules:

$\gamma \triangleq \langle\langle \texttt{if } \mathit{true} \texttt{ then x:=1 else skip} \rangle_{\mathsf{k}} \langle \mathtt{y} \mapsto 5 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$,

$\langle \gamma^s, \phi \rangle \triangleq \langle\langle \texttt{if } y^s >_{Int} 3 \texttt{ then x:=1 else skip} \rangle_{\mathsf{k}} \langle \mathtt{y} \mapsto y^s \rangle_{\mathsf{env}} \langle y^s >_{Int} 0 \rangle_{\mathsf{cnd}} \rangle_{\mathsf{cfg}}$,

$\langle \gamma'^s, \phi' \rangle \triangleq \langle\langle \texttt{skip} \rangle_{\mathsf{k}} \langle \mathtt{y} \mapsto y^s \rangle_{\mathsf{env}} \langle y^s >_{Int} 0 \wedge \neg(y^s >_{Int} 3) \rangle_{\mathsf{cnd}} \rangle_{\mathsf{cfg}}$,

$\alpha \triangleq \langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \cdots \rangle_{\mathsf{k}} \cdots \rangle_{\mathsf{cfg}} \wedge \neg B \Rightarrow \langle\langle S_2 \cdots \rangle_{\mathsf{k}} \cdots \rangle_{\mathsf{cfg}}$, and $\alpha^{sym} \triangleq$

$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \cdots \rangle_{\mathsf{k}} \langle \psi \rangle_{\mathsf{cnd}} \cdots \rangle_{\mathsf{cfg}} \wedge \neg \texttt{unsat}(\psi \wedge \neg B) \Rightarrow \langle\langle S_2 \cdots \rangle_{\mathsf{k}} \langle \psi \wedge \neg B \rangle_{\mathsf{cnd}} \cdots \rangle_{\mathsf{cfg}}$

Then, $\gamma \models \langle \gamma^s, \phi \rangle$ with $y^s \mapsto 5$, $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ since $\neg(y^s >_{Int} 3) \wedge y^s >_{Int} 0$ is satisfiable (e.g., $y^s \mapsto 2$), but the only transition starting from γ is $\gamma \overset{\alpha'}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \langle \mathtt{x := 1} \rangle_{\mathsf{k}} \langle \mathtt{y} \mapsto 5 \rangle_{\mathsf{env}}$, whose destination clearly does not satisfy $\langle \gamma'^s, \phi' \rangle$.

□

Thus, we need another way of proving the precision result. The next lemma says that the concrete semantics backwards-simulates the symbolic one:

Lemma 3 ($(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ **backward simulates** $(\mathcal{T}_{Cfg^{sym}}^{sym}, \Rightarrow_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}})$**)** *For all (symbolic) configurations* γ', $\langle \gamma^s, \phi \rangle$ *and* $\langle \gamma'^s, \phi' \rangle$, *if* $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ *and* $\gamma' \models \langle \gamma'^s, \phi' \rangle$ *then there exists* $\gamma \in \mathcal{T}_{Cfg}$ *such that* $\gamma \models \langle \gamma^s, \phi \rangle$ *and* $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$.

Proof The transition $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ is obtained by applying the rule $\alpha^{sym} \triangleq (\langle l, \psi \rangle \wedge \neg\, \texttt{unsat}(\psi \wedge b) \Rightarrow \langle r, \psi \wedge b \rangle)$. Thus, there are: a valuation $\rho^s : Var \to \mathcal{T}^{sym}$ such that $\rho^s(l) = \gamma^s$, $\rho^s \models \neg\, \texttt{unsat}(\psi \wedge b)$, $\rho^s(\psi) = \phi$, $\rho^s(\psi \wedge b) = \phi'$, $\rho^s(r) = \gamma'^s$; and a valuation $\vartheta : SymVal \to \mathcal{T}$ such that $\vartheta(\gamma'^s) = \gamma'$,$\vartheta \models \phi'$.

From the previous statement, we have $\phi' = \rho^s(\psi \wedge b) = \rho^s(\psi) \wedge \rho^s(b)$. Since $\vartheta \models \phi'$ we deduce $\vartheta \models \rho^s(\psi)$, thus, $\vartheta \models \phi$. Let us consider $\gamma \triangleq \vartheta(\rho^s(l)) = \vartheta(\gamma^s)$. The last two statements ensure $\gamma \models \langle \gamma^s, \phi \rangle$. There remains to prove $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$.

For this, consider the valuation $\rho \triangleq \vartheta \circ \rho^s$. From $\vartheta \models \phi'$ we obtain $\vartheta \models \rho^s(b)$, which is $(\vartheta \circ \rho^s)(b) = \mathit{true}$, i.e., $\rho(b) = \mathit{true}$. Finally, $\rho(r) = (\vartheta \circ \rho^s)(r) = \vartheta(\rho^s(r)) = \vartheta(\gamma'^s) = \gamma'$, which proves $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$ and completes the proof. □

The corollary to this lemma is called *weak precision*; it says that if a sequence β^{sym} of symbolic rules can be executed starting in some initial symbolic configuration *and the final symbolic configuration that the sequence reaches is satisfiable* (by some concrete configuration), then the corresponding sequence of concrete rules can be fired as well. A strong version of precision is proved after this one.

Corollary 3 (Weak Precision) *With the above notations,* $\langle \gamma^s, \phi \rangle \overset{\beta^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ *and* $\gamma' \models \langle \gamma'^s, \phi' \rangle$ *implies that there exists* γ *such that* $\gamma \overset{\beta}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$ *and* $\gamma \models \langle \gamma^s, \phi \rangle$.

Proof By induction on the length of β^{sym}, using Lemma 3 in the induction step. □

The strong version of precision (simply called precision), shown below, is based on the following lemma, which assumes the completeness of the `unsat` predicate.

Lemma 4 *If* $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ *then there are concrete configurations* γ, γ' *such that* $\gamma \models \langle \gamma^s, \phi \rangle$, $\gamma' \models \langle \gamma'^s, \phi' \rangle$ *and* $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$.

Proof The transition $\langle \gamma^s, \phi \rangle \overset{\alpha^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ is obtained by applying the rule $\alpha^{sym} \triangleq (\langle l, \psi \rangle \wedge \neg\, \texttt{unsat}(\psi \wedge b) \Rightarrow \langle r, \psi \wedge b \rangle)$. Thus, there is $\rho^s : Var \to \mathcal{T}^{sym}$ such that $\rho^s(l) = \gamma^s$, $\rho^s \models \neg\, \texttt{unsat}(\psi \wedge b)$, $\rho^s(\psi) = \phi$, $\rho^s(\psi \wedge b) = \phi'$, $\rho^s(r) = \gamma'^s$. From the above we get $\texttt{unsat}(\phi \wedge \rho^s(b)) = \mathit{false}$, and by completeness of `unsat`, $\phi \wedge \rho^s(b)$ is satisfiable, thus, there is $\vartheta : SymVal \to \mathcal{T}$ such that $\vartheta \models \phi \wedge \rho^s(b)$. With $\gamma \triangleq (\vartheta \circ \rho^s)(l)$, $\gamma' \triangleq (\vartheta \circ \rho^s)(r)$ we get $\gamma \models \langle \gamma^s, \phi \rangle$, $\gamma' \models \langle \gamma'^s, \phi' \rangle$, $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$. □

Corollary 4 (Precision) $\langle \gamma^s, \phi \rangle \overset{\beta^{sym}}{\Longrightarrow}_{\mathcal{S}^{sym}}^{\mathcal{T}^{sym}} \langle \gamma'^s, \phi' \rangle$ *implies that there exist concrete configurations* γ, γ' *such that* $\gamma \overset{\beta}{\Longrightarrow}_{\mathcal{S}}^{\mathcal{T}} \gamma'$ *and* $\gamma \models \langle \gamma^s, \phi \rangle$ *and* $\gamma' \models \langle \gamma'^s, \phi' \rangle$.

Proof By induction on the length of β^{sym},using Lemmas 3,4 in the induction step. □

The above precision corollary now captures the intuition of what precision means, informally: each symbolically executable path can also be executed concretely. It is essentially a theoretical result since it assumes an oracle to decide unsatisfiability in FOL, but it is nonetheless important since it shows that symbolic execution may only "diverge" from concrete execution only because of intrinsic undecidability results, (i.e., not because of how we have defined it in this paper).

6 Implementation

In this section present a prototype implementation of our approach in the $\mathbb{K}$ framework. We start with a short description of the support for symbolic artifacts already present in $\mathbb{K}$. Then, we show how a symbolic version of the IMP language (presented in Section 2) is obtained by applying the transformations from Section 4. Finally, we use resulting symbolic execution on the program shown in Figure 2 in order to prove that a certain property of it is violated. This illustrates the fact that symbolic execution is a good basis for developing symbolic analysers and verifiers grounded in the formal semantics of language.

6.1 Support for Symbolic Artifacts in the $\mathbb{K}$ Framework

For every sort, say, *S*, of a given $\mathbb{K}$ specification, the $\mathbb{K}$ tool provides users with a corresponding symbolic sort *SymbolicS*, together with a constructor `symS(` *Id* `)` for values in *SymbolicS*. For example, `symInt(a)` is a symbolic integer value, and `symBool(b)` is symbolic Boolean value.

The $\mathbb{K}$ framework also has an interface to the Z3 SMT solver, which consists of a function *checkSat* that takes as input a formula in a certain combination of theories (we currently use the theories for Booleans and integers, but more theories are available, for example, arrays). The function returns `sat` if the solver finds the formula satisfiable, `unsat` if it finds the formula unsatisfiable, and `unknown` if it cannot decide the (un)satisfiability.

6.2 Transforming IMP into IMP^{sym}

The IMP language, introduced in Section 2, is transformed into the symbolic language IMP^{sym} by following the transformation rules discussed in Section 4. We focus here on the transformation of the semantics (Section 4.3). The semantics of IMP language is shown in Figure 4 and is transformed as follows:

1. *Linearising rules.* The semantics of IMP has only two non-linear rules - the two last ones in Figure 4, which correspond to variable lookup and store. Their linearisation and the resulting rules were presented in Section 4.3.1.

2. *Replacing constants by variables.* This step concerns the rules that contain Boolean or integer constants in their left hand side, i.e., the following ones:

$$\langle\langle \mathit{true}\ \texttt{and}\ B\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle B\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \mathit{false}\ \texttt{and}\ B\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if}\ \mathit{true}\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if}\ \mathit{false}\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$

   Each occurrence of a constant is replaced by a fresh variable of the same sort (i.e., making sure that the linearity of the left-hand sides is preserved), and an equality is added to rule's condition. This generates the following rules:

$$\langle\langle B'\ \texttt{and}\ B\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge B' = \mathit{true} \Rightarrow \langle\langle B\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B'\ \texttt{and}\ B\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge B' = \mathit{false} \Rightarrow \langle\langle \mathit{false}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge B = \mathit{true} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge B = \mathit{false} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$

3. *Adding formulas to configuration and rules.* This step consists to modifying the configuration by adding a cell for path conditions. The original configuration structure of IMP, shown in Figure 3, becomes after this step:

$$\mathit{Cfg}^{sym} \quad ::= \langle\langle \mathit{Code}\rangle_{\mathsf{k}}\langle \mathit{Map}_{\mathit{Id},\mathit{Int}}\rangle_{\mathsf{env}}\langle \mathit{Fol}\rangle_{\mathsf{cnd}}\rangle_{\mathsf{cfg}}$$

   We proceed by modifying the semantic rules of IMP into symbolic rules, which have the form (2) (Page 11). We first note that unconditional rules - which can be written as $l \wedge b \Rightarrow r$ with $b \triangleq \mathit{true}$ - do not need to be transformed. Indeed, the transformation would generate the following rule:

$$\langle l, \psi\rangle \wedge \neg\, \texttt{unsat}(\psi) \Rightarrow \langle r, \psi\rangle \tag{3}$$

   Now, when such a rule is executed on a given symbolic configuration $\langle \gamma^s, \phi\rangle$:

   - if the rule (3) is the first one being executed then $\neg\texttt{unsat}(\phi) = \mathit{true}$ (since it does not make sense to start execution in unsatisfiable configurations);

- otherwise, a rule was applied earlier in the execution, ensuring $\neg\texttt{unsat}(\phi) = \mathit{true}$ (or else $\langle \gamma^s, \phi \rangle$ would not have been generated in the first place).

In both cases, $\neg\texttt{unsat}(\phi) = \mathit{true}$, hence, the transformation of our unconditional rule $l \Rightarrow r$ would also produce an unconditional rule $\langle l, \psi \rangle \Rightarrow \langle r, \psi \rangle$. And finally, users do not have to write the $\langle\rangle_{\mathsf{cnd}}$ cell (holding the ψ component in Cfg^{sym}) thanks to a nice feature of $\mathbb{K}$ called *configuration abstraction*, which, here, automatically completes rules operating over the configurations Cfg with the cells needed for operating over extended configurations Cfg^{sym}.

Thus, the only rules concerned by transformation are the conditional ones:

$$\langle\langle I_1 \texttt{ / } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1 /_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B' \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge B' = \mathit{true} \Rightarrow \langle\langle B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B' \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge B' = \mathit{false} \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge B = \mathit{true} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge B = \mathit{false} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

Every rule shown above is replaced by a rule of the form (2):

$$\langle\langle I_1 \texttt{ / } I_2 \cdots\rangle_{\mathsf{k}}\langle\psi\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}} \wedge \neg\texttt{unsat}(\psi \wedge I_2 \neq 0) \Rightarrow \langle\langle I_1 /_{Int} I_2 \cdots\rangle_{\mathsf{k}}\langle\psi \wedge I_2 \neq 0\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B' \texttt{ and } B \cdots\rangle_{\mathsf{k}}\langle\psi\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}} \wedge \neg\texttt{unsat}(\psi \wedge B' = \mathit{true}) \Rightarrow \langle\langle B \cdots\rangle_{\mathsf{k}}\langle\psi \wedge B' = \mathit{true}\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle B' \texttt{ and } B \cdots\rangle_{\mathsf{k}}\langle\psi\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}} \wedge \neg\texttt{unsat}(\psi \wedge B' = \mathit{false}) \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}}\langle\psi \wedge B' = \mathit{false}\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}}\langle\psi\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}} \wedge \neg\texttt{unsat}(\psi \wedge B = \mathit{true}) \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}}\langle\psi \wedge B = \mathit{true}\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}}\langle\psi\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}} \wedge \neg\texttt{unsat}(\psi \wedge B = \mathit{false}) \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}}\langle\psi \wedge B = \mathit{false}\rangle_{\mathsf{cnd}} \cdots\rangle_{\mathsf{cfg}}$$

6.3 Symbolic execution of IMP programs

We now illustrate the symbolic execution of the program shown in Figure 2. The program returns the wrong minimum when `a<=b` and `a>c`, as it sets `min` to `b` instead of `c`. We first extend the language with an `asssert` statement, which blocks execution if the Boolean expression given to is as argument does not hold.

We use symbolic execution to explore all possible paths of program from Figure 2 and check whether `min` is really less or equal than `a`, `b`, and `c` using `assert`. For this, we execute the following symbolic configuration (in ASCII $\mathbb{K}$):

```
<k>
if ( a <= b )
  then if (a <= c)
    then min := a
    else min := b
  else if (b <= c)
    then min := b
    else min := c;
assert(min <= a);
assert(min <= b);
assert(min <= c);
</k>
<env> a -> symInt(a), b -> symInt(b), c -> symInt(c) </env>
<cnd> true </cnd>
```

In the initial configuration the path condition is `true` and the variables `a`, `b`, `c` are mapped to the symbolic values `symInt(a)`, `symInt(b)`, `symInt(c)` respectively. When the execution reaches the first `if` statement, the $\mathbb{K}$ running tool (with the `search` option for full state-space exploration) generates both branches determined by the condition of the stamenent, which evaluates to *true* or *false* depending on whether `symInt(a)` $\leq$ `symInt(b)` or `symInt(a)` $>$ `symInt(b)`. The same process happens for the other `if` statements. When the program reaches `assert(min <= c)` on the execution branch that generated the path condition `symInt(a)` $\leq$ `symInt(b)` $\wedge$ `symInt(a) > symInt(c)` $\wedge$ `min = symInt(b)`, the execution gets stuck because the assertion contradicts the path condition. Moreover, the top of the `k` cell contains the failed assertion, which users can examine and compare with the path condition for debugging purposes. The complete IMP definition together with this example and instructions for running it are available at `https://fmse.info.uaic.ro/tools/Symbolic-IMP/`.

7 Conclusion and Future Work

We have presented a generic framework for the symbolic execution of programs in languages having operational semantics defined by term-rewriting. Starting from the formal definition of a language $\mathcal{L}$, the symbolic version $\mathcal{L}^{sym}$ of the language is automatically constructed, by extending (some of) the datatypes used in $\mathcal{L}$ with symbolic values, and by modifying the semantical rules of $\mathcal{L}$ in order to make them operate on symbolic values appropriately. The symbolic semantics of $\mathcal{L}$ is then the (usual) semantics of $\mathcal{L}^{sum}$, and symbolic execution of programs in $\mathcal{L}$ is the (usual) execution of the corresponding programs in $\mathcal{L}^{sym}$, which is the application of the rewrite rules of the semantics of $\mathcal{L}^{sym}$ to programs.

Assuming a sound unsatisfiability predicate for first-order logic, our symbolic execution has the expected properties of *coverage*, meaning that to each concrete execution there is a symbolic one that corresponds to it, and of *weak precision*, meaning that to each symbolic execution that ends in a satisfiable symbolic configuration has a concrete execution that corresponds to it. Here, correspondence means executing the same path in the control flow of the program. By assuming also a complete unsatisfiability predicate for first-order logic one also gets the theoretical *precision* result, meaning that each symbolic execution has a concrete execution that corresponds to it. The latter result is essentially theoretical since first-order logic is undecidable, but it means that any "imprecision" is not due to the way we defined symbolic simulation but to inherent undecidability results.

The results obtained are the expected ones; however, they were obtained thanks to carefully constructed definitions of what the essentials of a programming language are, in an algebraic and

term-rewriting based setting. The crucial observation that led us to the appropriate definitions is that datatypes are used by, but are not part of, a language definition, and thus should be treated differently. Finally, we have illustrated the framework on a simple imperative language defined in the $\mathbb{K}$ framework and have implemented a prototype of it also in $\mathbb{K}$.

Future Work We are planning to use symbolic execution as the basic mechanism on the deductive systems of program logics also developped in the $\mathbb{K}$ framework (such as reachability logic [21] and our own circular equivalence logic [14]) are built. More generally, our symbolic execution can be used for program testing, debugging, and verification, following the ideas presented in related work, but with the added value of being grounded in formal operational semantics.

References

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Yi [27], pages 52–68.

[3] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

[4] Jonathan de Halleux and Nikolai Tillmann. Parameterized unit testing with Pex. In *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 171–181. Springer, 2008.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[6] Laura K. Dillon. Verifying general safety properties of Ada tasking programs. *IEEE Trans. Softw. Eng.*, 16(1):51–63, January 1990.

[7] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electr. Notes Theor. Comput. Sci.*, 238(3):103–119, 2009.

[8] Pex: Automated exploratory testing for .NET. `http://research.microsoft.com/en-us/projects/pex`.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[10] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[11] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[12] Nam Hee Lee and Sung Deok Cha. Generating test sequences using symbolic execution for event-driven real-time systems. *Microprocessors and Microsystems*, 27(10):523 – 531, 2003.

[13] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 609–615. Springer, 2011.

[14] Dorel Lucanu and Vlad Rusu. Program equivalence by circular reasoning. Rapport de recherche RR-8116, INRIA, October 2012.

[15] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[16] José Meseguer. Rewriting logic and Maude: Concepts and applications. In Leo Bachmair, editor, *RTA*, volume 1833 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2000.

[17] José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.

[18] Corina S. Păsăreanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[19] Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors. *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM, 2010.

[20] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[21] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 555–574. ACM, 2012.

[22] Peter H. Schmitt and Benjamin Weiß. Inferring invariants by symbolic execution. In Bernhard Beckert, editor, *VERIFY*, volume 259 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

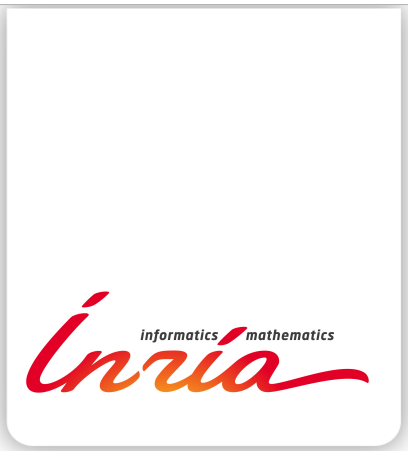
[23] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[24] Traian-Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

[25] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In Lori L. Pollock and Mauro Pezzè, editors, *ISSTA*, pages 157–168. ACM, 2006.

[26] Matt Staats and Corina S. Păsăreanu. Parallel symbolic execution for structural test generation. In Paolo Tonella and Alessandro Orso, editors, *ISSTA*, pages 183–194. ACM, 2010.

[27] Kwangkeun Yi, editor. *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*. Springer, 2005.



RESEARCH CENTRE
LILLE – NORD EUROPE

Parc scientifique de la Haute-Borne
40 avenue Halley - Bât A - Park Plaza
59650 Villeneuve d'Ascq

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399