Symbolic execution based on language transformation

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

Symbolic Execution based on Language Transformation

Andrei Arusoaie[b,*], Dorel Lucanu[a], Vlad Rusu[b]

[a] *Faculty of Computer Science, Alexandru Ioan Cuza University, Iaşi, Romania*
[b] *Inria Lille Nord Europe, France*

Abstract

We propose a language-independent symbolic execution framework for languages endowed with a formal operational semantics based on term rewriting. Starting from a given definition of a language, a new language definition is generated, with the same syntax as the original one, but whose semantical rules are transformed in order to rewrite over logical formulas denoting possibly infinite sets of program states. Then, the symbolic execution of concrete programs is, by definition, the execution of the same programs with the symbolic semantics. We prove that the symbolic execution thus defined has the properties naturally expected from it (with respect to concrete program execution). A prototype implementation of our approach was developed in the $\mathbb{K}$ Framework. We demonstrate the tool's genericity by instantiating it on several languages, and illustrate it on the reachability analysis and model checking of several programs.

Keywords: symbolic execution, formal semantics, programming languages, program analysis

1. Introduction

Symbolic execution is a well-known program analysis technique introduced in 1976 by James C. King [19]. Since then, it has proved its usefulness for testing, verifying, and debugging programs. Symbolic execution consists of executing programs with symbolic inputs, instead of concrete ones, and it involves the processing of expressions involving symbolic values [25]. 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 are diverse and include automated test input generation [20], [38], invariant detection [24], model checking [18], and proving program correctness [37, 12].

*Corresponding author
Email address: `andrei.arusoaie@inria.fr` (Andrei Arusoaie)

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 the 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 induced by the program instructions 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 conditional one, whose Boolean condition depends on symbolic values) then there is a branching in the tree. The path condition can then be used to distinguish between different branches.

Our contribution. The main contribution of the paper is a formal, language-independent theory and tool for symbolic execution, based on a language's operational semantics defined by term-rewriting[1]. On the theoretical side, we define a transformation of languages such that the symbolic execution of programs in the source language is, by definition, the concrete execution in the transformed language. We prove that the symbolic execution thus defined has the following properties, which relate it to concrete execution in a natural way:

Coverage: to every concrete execution there corresponds a feasible symbolic one;

Precision: to every feasible symbolic execution there corresponds a concrete one;

where two executions are said to be corresponding if they take the same path, and a symbolic execution is feasible if the path conditions along it are satisfiable. These theoretical properties have practical consequences, since they ensure that analyses based on symbolic program execution (reachability analysis, model checking, . . .) can be soundly transferred to concrete executions.

On the practical side, we present the prototype implementation of our approach in $\mathbb{K}$ [29] (version 3.4), a framework dedicated to defining formal operational semantics of languages. A $\mathbb{K}$ language definition is compiled into a Maude rewrite theory. Our prototype is based on several transformations which are encoded as compilation steps in the $\mathbb{K}$ definition compiler. The relationships between $\mathbb{K}$ language definitions and their compilation into Maude, and between the transformed $\mathbb{K}$ definitions and their Maude encodings are investigated in [4]. In this paper we briefly describe our implementation as a language-engineering tool, and demonstrate its genericity by instantiating it on nontrivial languages defined in $\mathbb{K}$.

We emphasise that the tool uses the $\mathbb{K}$ language-definitions as they are, without requiring modifications, and automatically harnesses them for symbolic execution. The examples illustrate reachability analysis, Linear Temporal Logic model checking, and bounded model checking using our tool.

The proposed approach uses a generic theoretical framework, which abstracts away details from the $\mathbb{K}$ implementation. In fact, $\mathbb{K}$ definitions are particular

[1]Most existing operational semantics styles (small-step, big-step, reduction with evaluation contexts, . . .) have been shown to be faithfully representable by rewriting [36].

examples of the abstract notion of language definition presented in Section 3.4.

A restriction of our approach is that it requires a clear distinction between code and data. We only deal with symbolic data (e.g., integers, booleans, etc.), but not with symbolic code. This excludes, for example, higher-order functional languages in which code can be passed as data between functions. The main goal of this work is to provide a language-independent symbolic execution framework on top of which different analysis tools can be developed (i.e., test case generators, program verification tools, etc.). This framework is intended to capture the central concepts of symbolic execution (e.g., symbolic values, path conditions, symbolic execution trees) which are completely independent of the chosen programming language. In practice, up to now, the framework has been successfully used for general purpose languages such as imperative, object-oriented, and scripting languages (as shown in Section 7.2), and for domain specific languages (e.g., OCL [3]).

Related work. There is a substantial number of tools performing symbolic execution available in the literature. However, most of them have been developed for specific programming languages and are based on informal semantics. Here we mention some of them that are strongly related to our approach.

Java PathFinder [26] 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 several Satisfiability Modulo Theories (SMT) solvers and the user can also choose between multiple decision procedures. We anticipate that by instantiating our generic approach to a formal definition of Java (currently being defined in the $\mathbb{K}$ framework) we obtain some of Java PathFinder's features for free.

Another approach consists in combining concrete and symbolic execution, also known as *concolic* execution. First, some concrete values given as input 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: DART [16], CUTE [34], EXE [8], PEX [10]. We note that our approach allows mixed concrete/symbolic execution; it can be the basis for language-independent implementations of concolic execution.

Symbolic execution has initially been used in automated test generation [19]. It can also be used for proving program correctness. There are several tools (e.g. Smallfoot [6, 39]) which use symbolic execution together with separation logic to prove Hoare triples. There are also approaches that attempt to automatically detect invariants in programs([24], [33]). Another useful application of symbolic execution is the static detection of runtime errors. The main idea is to perform symbolic execution on a program until a state is reached where an error occurs, e.g., null-pointer dereference or division by zero. We show that the implementation prototype we developed is also suitable for such static code analyses.

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 has sucessfully been applied to the analysis of security protocols [23]. Such analyses rely on powerful unification-modulo-theories algorithms [14], which work well for security protocols since there are unification algorithms modulo the theories involved there (exclusive-or, . . .). This is not always the case for programming languages with arbitrarily complex datatypes. *Rewriting modulo SMT* [27] is a recently introduced technique for performing symbolic execution on rewrite theories. Their approach and ours have some common features: a built-in subtheory (for *data*, in our case) in which constraints are handled by SMT solving; the notion of constrained terms (in our case, *Matching Logic patterns*); and soundness and completeness results (in our case, *precision and coverage*). The main difference is that they focus on rewriting-logic specifications, whereas we focus on language definitions.

The present paper is an extended version of our SLE 2013 paper [2]. It relies on a more general way of defining programming languages, consisting in using (Topmost) Matching Logic to denote sets of program states and Reachability Logic for the operational semantics of languages. The two logics are briefly introduced in the paper; for details readers can consult [30]. This results in better definitions for essential notions such as *symbolic domain* (the domain over which symbolic execution "computes"). The new definitions are more suitable because they faithfully capture the essence of what symbolic execution is about: computing with logical constraints denoting sets of program states. By contrast, in [2] the corresponding definitions were purely axiomatic: they required certain abstract diagrams to be commutative. The new approach also extends the range of language definitions for which symbolic execution can be defined: the previous approach [2] is now an instance of the current one. Among the extensions we mention axiomatically-defined structures (sets, bags, lists, ...), with axioms such as associativity, commutativity, unity and combinations thereof. Such structures are intensively used in real-life language definitions in the $\mathbb{K}$ framework (C [13], Java[7], . . .). Finally, a technical improvement is that our new definition of the symbolic transition relation does not distinguish among semantically equivalent symbolic states.

Structure of the paper. Section 2 introduces our running example (the imperative language IMP) and its definition in $\mathbb{K}$.

Section 3 introduces a framework for language definitions, making our approach generic in both the language-definition framework and the language being defined; $\mathbb{K}$ and IMP are just instances for the former and latter, respectively.

Section 4 introduces the notion of *symbolic domain*, which formalises the domain in which symbolic execution takes place.

Section 5 then shows how the definition of a language $\mathcal{L}$ can be transformed into the definition of a language $\mathcal{L}^{\mathfrak{s}}$ by replacing the concrete domain with the symbolic one, and by providing the semantical rules of $\mathcal{L}$ with means to operate in the new, symbolic domain. The coverage and precision results are proved.

Section 6 then gives two instances of the framework introduced in the previous section. The first one is isomorphic to that presented in [2], whereas the second strictly generalizes the first one by including axiomatically-defined structures which, as previously mentioned, are intensively used in real-life language definitions.

Section 7 describes an implementation of our approach in the $\mathbb{K}$ framework and shows how it is automatically instantiated to nontrivial languages defined in $\mathbb{K}$. Applications to program analysis are given.

Section 8 concludes and discusses future work.

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 (e.g., [31, 28]). The syntax of IMP is described in Figure 1 and is mostly self-explanatory 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.

$$
\begin{array}{rl}
\mathit{Id} ::= & \text{domain of identifiers} \\
\mathit{Int} ::= & \text{domain of integer numbers (including operations)} \\
\mathit{Bool} ::= & \text{domain of boolean constants (including operations)} \\
\mathit{AExp} ::= & \mathit{Int} \quad \mid \mathit{AExp}\ \texttt{/}\ \mathit{AExp}\ [\text{strict}] \\
& \mid \mathit{Id} \quad \mid \mathit{AExp}\ \texttt{*}\ \mathit{AExp}\ [\text{strict}] \\
& \mid (\mathit{AExp}) \quad \mid \mathit{AExp}\ \texttt{+}\ \mathit{AExp}\ [\text{strict}] \\
\mathit{BExp} ::= & \mathit{Bool} \\
& \mid (\mathit{BExp}) \quad \mid \mathit{AExp}\ \texttt{<=}\ \mathit{AExp}\ [\text{strict}] \\
& \mid \texttt{not}\ \mathit{BExp}\ [\text{strict}] \quad \mid \mathit{BExp}\ \texttt{and}\ \mathit{BExp}\ [\text{strict(1)}] \\
\mathit{Stmt} ::= & \texttt{skip} \mid \texttt{\{}\ \mathit{Stmt}\ \texttt{\}} \mid \mathit{Stmt}\ \texttt{;}\ \mathit{Stmt} \mid \mathit{Id}\ \texttt{=}\ \mathit{AExp} \\
& \mid \texttt{while}\ \mathit{BExp}\ \texttt{do}\ \mathit{Stmt} \\
& \mid \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}\ [\text{strict(1)}] \\
\mathit{Code} ::= & \mathit{Id} \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{AExp} \mid \mathit{BExp} \mid \mathit{Stmt} \mid \mathit{Code} \curvearrowright \mathit{Code}
\end{array}
$$

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

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

Figure 2: $\mathbb{K}$ Configuration of IMP

The operational semantics of IMP is given as a set of (possibly conditional) rewrite rules. The terms to which rules are applied are called *configurations*. Configurations typically contain the program to be executed, together with any

$$\langle\langle I_1 \texttt{ + } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ * } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 *_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle I_1 \texttt{ / } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 /_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} I_2 \neq_{Int} 0$$
$$\langle\langle I_1 \texttt{ <= } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 \leq_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \mathit{true} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \mathit{false} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{not } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \neg B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{skip } \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle S_1 ; S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \curvearrowright S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{\{ } S \texttt{ \}} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } \mathit{true} \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{if } \mathit{false} \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle \texttt{while } B \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow$$
$$\langle\langle \texttt{if } B \texttt{ then\{ } S \texttt{ ;while } B \texttt{ do } S \texttt{ \}else skip} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$
$$\langle\langle X \cdots\rangle_{\mathsf{k}} \langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{lookup}(X, E) \cdots\rangle_{\mathsf{k}} \langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$
$$\langle\langle X \texttt{ = } I \cdots\rangle_{\mathsf{k}} \langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \langle \mathit{update}(X, E, I)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

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

additional information required for program execution. The structure of a configuration depends on 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 this consists of a top cell `cfg`, having a subcell `k` containing the code and a subcell `env` containing the environment (cf. Figure 2). 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. Computation tasks are typically statements and expressions. The environment in the `env` cell is a multiset of bindings of identifiers to values, e.g., $\texttt{a} \mapsto 3, \texttt{b} \mapsto 1$.

The semantics of IMP is shown in Figure 3. 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 syntactical constructions require only 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 3 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 Boolean expressions, B ranges over the Boolean values

$\{false, true\}$, 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. Language Definitions

Creating a language-independent symbolic execution framework requires *specifications* of programming languages, which, for each language, define the meanings of the programs in that language. The idea behind our approach is to pass such specifications as parameters to our symbolic execution framework.

This section presents some background used in the rest of the paper. We start by giving a brief description of the basics of algebraic specifications, of many-sorted First Order Logic (FOL), and of Matching Logic (ML) [30], together with the notation and conventions that we are going to use throughout the paper. Then, we present the general notion of language definition using Matching Logic and Reachability Logic (RL) [30]. The syntax of programming languages and the data types used in their semantics are given using algebraic specifications, while their semantics is given using RL.

3.1. Algebraic Specifications

In this section we briefly introduce some basic definitions and notations regarding algebraic specifications that we use in the paper.

The use of algebraic specifications to model computer programs is motivated by the fact that they can manipulate several kinds of *sorts* of data, in the same way as programs do. For instance, the name *Int* is a sort and it is part of the syntax. The syntax is called *signature*, and it consists of a set of sorts. Formally, given S a set of sorts, an *S-sorted signature* Σ is an $S^* \times S$-indexed family of sets $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ of sets whose elements are called *operation symbols*. If $S' \subseteq S$, an S'-sorted signature Σ' is a *subsignature* of an S-sorted signature Σ if $\Sigma' \subseteq \Sigma$ as $S^* \times S$-indexed sets.

The BNF syntax of IMP (Figure 1) has a corresponding S_{IMP}-sorted signature Σ_{IMP}. Nonterminals in the grammar (e.g. *Int*, *Bool*, *AExp*, etc.) are sorts in S_{IMP}, while each grammar production has a corresponding operation symbol in Σ_{IMP}. For instance, the production $AExp ::= AExp$ + $AExp$, has a corresponding operation symbol `_+_` : $AExp \times AExp \to AExp$ having two arguments of sort $AExp$ and result of sort $AExp$. An operational symbols without arguments is called *constant*; e.g., *true* and *false* are constants of sort *Bool*.

The meaning of algebraic signatures Σ is given by Σ-algebras, called also Σ-models. A *Σ-algebra M* consists of an S-indexed set (also denoted M), i.e., a *carrier set* M_s for each sort $s \in S$; an element $M_c \in M_s$ interpreting each constant symbol c as an actual element; and a function $M_f : M_{s_1} \times \ldots M_{s_n} \to M_s$ interpreting each operation symbol f as a function. The interpretation of a constant can be seen as a particular (constant) function.

An example of Σ-algebra M for IMP, used in this paper, interprets the sort *Int* by the set M_{Int} of integers, the sort *Bool* by the set of booleans $M_{Bool} = \{false, true\}$, and the other nonterminals by their corresponding syntactical categories, e.g. M_{AExp} is the set of arithmetical expressions. M also interprets all operation symbols by functions, i.e. the symbol $\_ +_{Int} \_$ is interpreted by a function $M_{\_+_{Int}\_} : M_{Int} \times M_{Int} \to M_{Int}$ which is the addition over integers, and `_+_` is interpreted by a function $M_{\_+\_} : M_{AExp} \times M_{AExp} \to M_{AExp}$ which is the expression constructor.

Let Var be an S-indexed set of *variables*. The S-indexed set $T_\Sigma(Var) = \{T_{\Sigma,s}(Var) \mid s \in S\}$ of Σ-*terms* t is defined by:

$$t ::= c \mid x \mid f(t, \ldots, t),$$

where c is a constant, $x \in Var$, and f is an operation symbol with n arguments. The *term* Σ-*algebra* $T_\Sigma(Var)$ has Σ-terms $T_\Sigma(Var)$ as carrier sets and interprets each constant symbol c by itself and each operation symbol $f : s_1 \ldots s_n \to s$ by the term constructor function $T_{s_1} \times \ldots \times T_{s_n} \to T_s$ that maps $(t_1, \ldots, t_n)$ into the term $f(t_1, \ldots, t_n)$. If Var is $\emptyset$, then $T_\Sigma(\emptyset)$ is the algebra of ground terms (terms without variables), which we denote it by T_Σ.

Examples of Σ terms are `_+_` $(2, 3)$, and `_*_` (`_+_` $(2, 3), x)$, where $x \in Var_{AExp}$. We often use the mixfix notation for terms, e.g. the above ones are written as `2 + 3`, respectively, `(2 + 3) *` x.

A *valuation* ρ is a function $\rho : Var \to M$ that maps variables to values from a Σ-model M. A *substitution* is a mapping $\sigma : X \to T_\Sigma(Var)$ for some $X \subseteq Var$. We often denote by σ (resp. ρ) the homomorphic extension of the substitution σ (resp. the valuation ρ) to terms. The composition of substitutions, and of a valuation and a substitution, is denoted by $\circ$ and coincides with the standard notion of function composition. The restriction of a valuation $\rho : Var \to M$ to a subset $X \subseteq Var$ is denoted by $\rho|_X$ and coincides with the standard notion of function restriction. Note that if M is the algebra of terms, then $\rho|_X$ is a substitution. Domains and ranges of functions are denoted as usual, i.e., for $f : A \to B$ we have $dom(f) = A$, $ran(f) = B$. A *congruence* $\cong$ (over terms) is an equivalence relation, which is compatible with the operations, i.e., for every operation f with n arguments, and arguments $t_1, \ldots, t_n, t'_1, \ldots, t'_n$, if $t_i \cong t'_i$ for all i, then $f(t_1, \ldots, t_n) \cong f(t'_1, \ldots, t'_n)$.

*3.2. Many-sorted First Order Logic (*FOL*)*

Given a set S of sorts, an S-sorted *first order signature* Φ is a pair (Σ, Π), where Σ is an algebraic S-sorted signature and Π is an indexed set of the form $\{\Pi_w \mid w \in S^*\}$ whose elements are called *predicate symbols*, where $p \in \Pi_w$ is said to have *arity* w. A Φ-*model* consists of a Σ-algebra M together with a subset $M_p \subseteq M_{s_1} \times \cdots \times M_{s_n}$ for each predicate $p \in \Pi_w$, where $w = s_1 \ldots s_n$. For instance, we may define $<_{Int}$ as a predicate symbol in $\Pi_{Int,Int}$ interpreted by the set of integer pairs (a, b) with a less than b (i.e., $a <_{Int} b$).

We now define the syntax of FOL formulas over a first order signature $\Phi = (\Sigma, \Pi)$ and a possibly infinite set of variables Var. Given a FOL signature $\Phi = (\Sigma, \Pi)$, the set of Φ-*formulas* is defined by

$$\phi ::= \top \mid p(t_1, \ldots, t_n) \mid \neg\phi \mid \phi \wedge \phi \mid (\exists V)\phi$$

where p ranges over predicate symbols Π, each t_i ranges over $T_\Sigma(\mathit{Var})$ of appropriate sort, and V over finite subsets of Var.

Given a first order Φ-model M,a Φ-formula ϕ, V a set of S-sorted variables, and a valuation $\rho : \mathit{Var} \to M$, the satisfaction relation $\rho \models \phi$ is defined as follows:

1. $\rho \models \top$;
2. $\rho \models p(t_1, \ldots, t_n)$ iff $(\rho(t_1), \ldots, \rho(t_n)) \in M_p$;
3. $\rho \models \neg\phi$ iff $\rho \models \phi$ does not hold;
4. $\rho \models \phi_1 \wedge \phi_2$ iff $\rho \models \phi_1$ and $\rho \models \phi_2$;
5. $\rho \models (\exists V)\phi$ iff there is $\rho' : \mathit{Var} \to M$ with $\rho'(x) = \rho(x)$ for all $x \notin V$, such that $\rho' \models \phi$.

A formula ϕ is *valid in* M, denoted by $M \models \phi$, if it is satisfied by all valuations ρ.

Let $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$ be a subsignature of (Σ, Π) and M be a (Σ, Π)-model. Then $M\upharpoonright_{(\Sigma^\mathfrak{d},\Pi^\mathfrak{d})}$ is the $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$-model $M^\mathfrak{d}$ defined as follows:

- $M^\mathfrak{d}_s = M_s$ for each $\Sigma^\mathfrak{d}$-sort;
- $M^\mathfrak{d}_f = M_f$ for each functional symbol f in $\Sigma^\mathfrak{d}$;
- $M^\mathfrak{d}_p = M_p$ for each predicate symbol p in $\Pi^\mathfrak{d}$.

The above definition is a particular case of *reduct model defined via a signature morphism* [32]; here, the morphism is given by inclusion. We use it in order to relate the model of data $M^\mathfrak{d}$ to that used in the semantics of programming languages.

3.3. Matching Logic (ML) and Reachability Logic (RL)

We recall from [30] the (topmost) ML and RL concepts and results used in this paper. First, we present the definition of ML formulas and the corresponding satisfaction relation, and then, the definition of RL formulas and the transition system generated by a set of RL formulas.

Definition 1 (ML Formula). An ML *signature* $\Phi = (\Sigma, \Pi, \mathit{Cfg})$ is a first-order signature (Σ, Π) together with a distinguished sort Cfg for *states*. The set of ML-*formulas* over Φ is defined by

$$\varphi ::= \pi \mid \top \mid p(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid (\exists V)\varphi$$

where the *basic pattern* π ranges over $T_{\Sigma,\mathit{Cfg}}(\mathit{Var})$, p ranges over predicate symbols Π, each t_i ranges over $T_\Sigma(\mathit{Var})$ of appropriate sorts, and V over finite subsets of Var. The sort Cfg is intended to model program states. We often call the ML formulas *patterns*.

The free occurrences of variables in ML formulas is defined as usual (i.e., like in FOL) and we let $\mathit{var}(\varphi)$ denote the set of variables freely occurring in φ.

Example 1. Let $\varphi \triangleq (\exists Z)\langle\langle \mathtt{x = y;\ skip}\rangle_{\mathsf{k}}\langle \mathtt{x}{\mapsto}X\ \mathtt{y}{\mapsto}Y\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (X \leq Z \wedge Z < Y)$. We have $var(\varphi) = \{X, Y\}$. Program variables $\mathtt{x}$, $\mathtt{y}$ should not be confused with logical variables X, Y; program variables are constants of sort Id.

Excepting the basic patterns, the semantics of ML formulas is similar to that of FOL ones:

Definition 2 (ML satisfaction relation). Given $\Phi = (\Sigma, \Pi, \mathit{Cfg})$ an ML signature, M a (Σ, Π)-model, φ an ML formula, $\gamma \in M_{\mathit{Cfg}}$ a state, V a (S-sorted) set of variables, and $\rho : \mathit{Var} \to M$ a valuation, the satisfaction relation $(\gamma, \rho) \models \varphi$ is defined as follows:

1. $(\gamma, \rho) \models \pi$ iff $\rho(\pi) = \gamma$;
2. $(\gamma, \rho) \models \top$;
3. $(\gamma, \rho) \models p(t_1, \ldots, t_n)$ iff $var(t_1, \ldots, t_n) \subseteq X$ and $(\rho(t_1), \ldots, \rho(t_n)) \in M_p$;
4. $(\gamma, \rho) \models \neg\varphi$ iff $(\gamma, \rho) \models \varphi$ does not hold;
5. $(\gamma, \rho) \models \varphi_1 \wedge \varphi_2$ iff $(\gamma, \rho) \models \varphi_1$ and $(\gamma, \rho) \models \varphi_2$; and
6. $\rho \models (\exists V)\phi$ iff there is $\rho' : \mathit{Var} \to M$ with $\rho'(x) = \rho(x)$, for all $x \notin V$, such that $\rho' \models \phi$.

The denotational semantics of an ML formula consists of all concrete configurations that match it:

Definition 3. Let φ be an ML formula. Then $[\![\varphi]\!]$ denotes the set of configurations $\{\gamma \mid (\exists\rho)(\gamma, \rho) \models \varphi\}$.

Example 2. Let $\varphi \triangleq \langle\langle \mathtt{x = y;\ skip}\rangle_{\mathsf{k}}\langle \mathtt{x}{\mapsto}X\ \mathtt{y}{\mapsto}Y\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (X \geq 0 \wedge Y \leq X)$ and $\gamma \triangleq \langle\langle \mathtt{x = y;\ skip}\rangle_{\mathsf{k}}\langle \mathtt{x}{\mapsto}7\ \mathtt{y}{\mapsto}3\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. Then $\gamma \in [\![\varphi]\!]$, since there exists ρ, with $\rho(X) = 7$ and $\rho(Y) = 3$, such that $(\gamma, \rho) \models \varphi$.

We now recall the definition of RL formulas, which are pairs of ML formulas, and of the transition system induced by a set of RL formulas. We consider a fixed ML signature $\Phi = (\Sigma, \Pi, \mathit{Cfg})$, a set of variables Var, and a fixed Φ-model M.

Definition 4 (RL Formula, RL System). An RL formula is a pair $\varphi \Rightarrow \varphi'$ of ML formulas. An RL *system* is a set $\mathcal{S}$ of RL formulas. The *transition system* defined by $\mathcal{S}$ over M is $(M_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}})$, where $\Rightarrow_{\mathcal{S}} = \{(\gamma, \gamma') \mid (\exists \varphi \Rightarrow \varphi' \in \mathcal{S})(\exists\rho)(\gamma, \rho) \models \varphi \wedge (\gamma', \rho) \models \varphi'\}$. We write $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ for $(\gamma, \gamma') \in \Rightarrow_{\mathcal{S}}$.

Example 3. Let $\varphi \triangleq \langle\langle \mathtt{x = y;\ skip}\rangle_{\mathsf{k}}\langle \mathtt{x}{\mapsto}X\ \mathtt{y}{\mapsto}Y\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (X \geq 0 \wedge Y \leq X)$, $\varphi' \triangleq \langle\langle \mathtt{skip}\rangle_{\mathsf{k}}\langle \mathtt{x}{\mapsto}Y\ \mathtt{y}{\mapsto}Y\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (X \geq 0 \wedge Y \leq X)$, and assume $\varphi \Rightarrow \varphi' \in \mathcal{S}$. Let $\gamma \triangleq \langle\langle \mathtt{x = y;\ skip}\rangle_{\mathsf{k}}\langle \mathtt{x}{\mapsto}7\ \mathtt{y}{\mapsto}3\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\gamma' \triangleq \langle \mathtt{skip}\rangle_{\mathsf{cfg}}\langle \mathtt{x}{\mapsto}3\ \mathtt{y}{\mapsto}3\rangle_{\mathsf{env}}$. Then, $(\gamma, \gamma') \in \Rightarrow_{\mathcal{S}}$, using the same valuation ρ as in Example 2.

3.4. Language Definitions

In this section we present the abstract notion of *language definition* used in the rest of the paper. $\mathbb{K}$ language definitions are particular examples of that. In a nutshell, a language definition consists of an ML signature Φ (including the syntax of the language, the configuration, ...), a model for Φ, and a set of RL formulas for the semantics.

Definition 5. A *language definition* is a tuple $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), M, \mathcal{S})$ where:

- $(\Sigma, \Pi, \mathit{Cfg})$ is a ML signature,
- M is a model of $(\Sigma, \Pi, \mathit{Cfg})$,
- $\mathcal{S}$ is a finite set of RL formulas, of the form $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$, where the ML formulas $\pi_1 \wedge \phi_1, \pi_2 \wedge \phi_2$ are over the signature $(\Sigma, \Pi, \mathit{Cfg})$.

We emphasise that the model M is part of a language definition. It may includes operations over primitive data types as integers and booleans, and data structures and their operations used to represent semantical ingredients.

In the following, we assume a (strict) subsignature $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ of (Σ, Π) for the language's data types (integers, lists, etc) and a $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$-model $\mathcal{D}$ such that M restricted to $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ equals $\mathcal{D}$, i.e., $M\upharpoonright_{(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})} = \mathcal{D}$. The sort Cfg is not a data sort. We sometimes call language definitions *languages* for simplicity. A language definition $\mathcal{L}$ induces a transition system $(M_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}})$, where $\Rightarrow_{\mathcal{S}}$ is given by Definition 4. The next example illustrates all these concepts on IMP.

Example 4. In the case of IMP, nonterminals in the syntax $(\mathit{Id}, \mathit{Int}, \mathit{Bool}, \ldots)$ are sorts in Σ. Each production from the syntax defines an operation in Σ; e.g, the production $\mathit{AExp} ::= \mathit{AExp}$ `+` AExp defines the operation `_+_` $: \mathit{AExp} \times \mathit{AExp} \to \mathit{AExp}$. These operations define the constructors of the result sort. For the sort Cfg, the only constructor is $\langle\langle \_ \rangle_{\mathsf{k}} \langle \_ \rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : \mathit{Code} \times \mathit{Map}_{\mathit{Id},\mathit{Int}} \to \mathit{Cfg}$. The expression $\langle\langle X = I \curvearrowright C\rangle_{\mathsf{k}} \langle X \mapsto 0\ \mathit{Env}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a term of $T_{\mathit{Cfg}}(\mathit{Var})$, 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 $\mathit{Map}_{\mathit{Id},\mathit{Int}}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets Int as the set of integers, the operations like $+_{\mathit{Int}}$ (cf. Figure 3) as the corresponding usual operation on integers, Bool as the set of Boolean values $\{\mathit{false}, \mathit{true}\}$, the operation like $\wedge$ as the usual Boolean operations, the sort $\mathit{Map}_{\mathit{Id},\mathit{Int}}$ as the multiset of maps $X \mapsto I$, where X ranges over identifiers Id and I over the integers Int. The value of an identifier X in an environment E is obtained by calling $\mathit{lookup}(X, E)$, and it is updated by calling $\mathit{update}(X, E, I)$. Here, $\mathit{lookup}()$ and $\mathit{update}()$ are operations in $\Sigma^{\mathfrak{d}}$. The other sorts, AExp, BExp, Stmt, and Code, are interpreted in the algebra M as ground terms over the signature Σ, in which data subterms are replaced by their interpretations in $\mathcal{D}$. For instance, the term `if` $1 >_{\mathit{Int}} 0$ `then skip else skip` of sort Stmt is intepreted as `if` $\mathcal{D}_{\mathit{true}}$ `then skip else skip`.

4. Symbolic Domain

This section is dedicated to defining the *symbolic domain*, in which symbolic execution takes place. Intuitively, symbolic execution deals with (possibly infinite) sets of concrete configurations, denoted by ML formulas. Since (possibly, infinitely) many ML formulas may denote the same set of concrete configurations, we shall be working with the following equivalence relation on ML formulas:

Definition 6 (Equivalence Relation on ML Formulas). Let φ and φ' be two ML formulas. Then $\varphi \sim \varphi'$ iff $[\![\varphi]\!] = [\![\varphi']\!]$.

Note that $\varphi \sim \varphi'$ does not imply, in general, $M \models \varphi \leftrightarrow \varphi'$. For instance, consider patterns $\varphi \triangleq \langle\langle \mathtt{x = } A +_{Int} 1\rangle_{\mathsf{k}}\langle \mathtt{x} \mapsto B +_{Int} 5\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A <_{Int} B$ and $\varphi' \triangleq \langle\langle \mathtt{x = } A'\rangle_{\mathsf{k}}\langle \mathtt{x} \mapsto B'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (A' =_{Int} A +_{Int} 1 \wedge B' =_{Int} B +_{Int} 5 \wedge A <_{Int} B$. We can easily observe that $\varphi \sim \varphi'$. However, $M \not\models \varphi \leftrightarrow \varphi'$: if $\gamma \triangleq \langle\langle \mathtt{x = } 3\rangle_{\mathsf{k}}\langle \mathtt{x} \mapsto 8\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\rho(A) = 2$, $\rho(B) = 3$, then $(\gamma, \rho) \models \varphi$ but $(\gamma, \rho) \not\models \varphi'$ because $\rho(A')$ can be different from 3 and/or $\rho(B')$ can be different from 8.

For symbolic execution we shall be needing to *unify* equivalence classes. This notion of unifier builds upon a standard notion of unification for terms. Hereafter we consider a given congruence relation $\cong$ on $T_\Sigma(Var)$.

Definition 7 (Unifier modulo congruence). A $\cong$-*unifier* of two terms t_1, t_2 is a substitution $\sigma : var(t_1, t_2) \to T_\Sigma(Var)$ such that $\sigma(t_1) \cong \sigma(t_2)$.

A set of $\cong$-unifiers for two terms is *complete* if every valuation that equates the two terms is an instance of at least one substitution in the set, and a $\cong$-unification algorithm computes completes sets of $\cong$-unifiers for its inputs:

Definition 8 (Complete set of unifiers). A set S of $\cong$-unifiers of two terms t_1, t_2 is *complete* if for each valuation $\rho : Var \to M$ such that $\rho(t_1) = \rho(t_2)$, there exists $\sigma \in S$ and $\eta : Var \to M$ such that $\rho|_{dom(\sigma)} = \eta \circ \sigma$.

The existence of unification algorithms is essential in the definition of the symbolic execution:

Definition 9. A $\cong$-*unification algorithm* is a function that takes two terms and returns a finite (possibly empty) and complete set of $\cong$-unifiers for the terms.

Since the unification modulo a congruence is undecidable in general, we work under the following assumption:

Assumption 1. *We assume a congruence $\cong$ on $T_\Sigma(Var)$ such that for all $t_1, t_2 \in T_\Sigma(Var)$, if $t_1 \cong t_2$ then $\rho(t_1) = \rho(t_2)$ for all valuations ρ.*

We also assume a $\cong$-unification algorithm, hereafter denoted by $unif_\cong(\_,\_)$.

Unification is now extended to equivalence classes of ML formulas of the form $\varphi \triangleq \pi \wedge \phi$, $\varphi' \triangleq \pi' \wedge \phi'$. The extension consists in considering ML formulas $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ and $(\exists X')\widetilde{\pi}' \wedge \widetilde{\phi}'$ that are ML-equivalent to φ and φ', respectively, such that $unif_\cong(\widetilde{\pi}, \widetilde{\pi}')$ is nonempty; and to define symbolic unifiers as follows:

Definition 10 (Unification on Equivalence Classes). An *abstraction* of the pattern $\pi \wedge \phi$ is a pattern $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ with the property that $M \models \pi \wedge \phi \leftrightarrow (\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$. Given abstractions $\widetilde{\varphi} \triangleq (\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ and $\widetilde{\varphi}' \triangleq (\exists X)\widetilde{\pi}' \wedge \widetilde{\phi}'$ of φ and φ', respectively, a *symbolic unifier* of the equivalences classes $[\varphi]_\sim, [\varphi']_\sim$ is the FOL formula

$$(\exists X \cup X' \cup var(ran(\sigma)))\phi^\sigma \wedge \widetilde{\phi} \wedge \widetilde{\phi}'$$

where $\sigma \in unif_{\cong}(\widetilde{\pi}, \widetilde{\pi}')$ and ϕ^σ denotes the FOL formula $\bigwedge_{x \in dom(\sigma)}(x = \sigma(x))$.

The set of *symbolic unifiers* of $[\varphi]_\sim$ and $[\varphi']_\sim$ is denoted by $unif([\varphi]_\sim, [\varphi']_\sim)$.

Unlike unifiers of terms, which are substitutions σ, unifiers of equivalence classes are FOL formulas that have ϕ^σ as a subformula. Intuitively, ϕ^σ plays for equivalence classes of ML formulas the role that σ plays for terms. The following sequence of implications/equivalences formalises this observation:

$$\begin{array}{ll}
\sigma(\pi) \cong \sigma(\pi') & \longrightarrow \\
\sigma(\pi) \sim \sigma(\pi') & \Longleftrightarrow \\
\pi \wedge \phi^\sigma \sim \pi' \wedge \phi^\sigma & \longrightarrow \\
\pi \wedge \phi^\sigma \wedge \widetilde{\phi} \wedge \widetilde{\phi}' \sim \pi' \wedge \phi^\sigma \wedge \widetilde{\phi} \wedge \widetilde{\phi}' &
\end{array}$$

Example 5. Consider the following patterns:

$$\pi \wedge \phi = \langle\langle I \texttt{ + } 3\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I >_{Int} 0,$$
$$\widetilde{\pi} \wedge \widetilde{\phi} = \langle\langle I \texttt{ + } J\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (I >_{Int} 0 \wedge J =_{Int} 3).$$

Then $M \models \pi \wedge \phi \leftrightarrow (\exists J)\widetilde{\pi} \wedge \widetilde{\phi}$, where M is the model for IMP (cf. Example 4). Consider also

$$\pi' \wedge \phi' \triangleq \langle\langle (A +_{Int} A) \texttt{ + } (B +_{Int} 1)\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A >_{Int} 0,$$
$$\widetilde{\pi}' \wedge \widetilde{\phi}' \triangleq \langle\langle A' \texttt{ + } B'\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (A' =_{Int} A +_{Int} A \wedge A >_{Int} 0 \wedge B' =_{Int} B +_{Int} 1).$$

For $\cong$ being the syntactical equality, we have $unif_{\cong}(\pi, \pi') = \emptyset$ but $unif_{\cong}(\widetilde{\pi}, \widetilde{\pi}') \neq \emptyset$. In particular, $\sigma = \{I \mapsto A', J \mapsto B'\} \in unif_{\cong}(\widetilde{\pi}, \widetilde{\pi}')$ and a symbolic unifier $\psi \in unif([\pi \wedge \phi]_\sim, [\pi' \wedge \phi']_\sim)$ is $(I =_{Int} A' \wedge J =_{Int} B') \wedge (I >_{Int} 0 \wedge J =_{Int} 3) \wedge (A' =_{Int} A +_{Int} A \wedge A >_{Int} 0 \wedge B' =_{Int} B +_{Int} 1)$.

Remark 1. Example 5 emphasises the role of pattern-abstractions for overcoming some problems raised by unification. During symbolic execution, programs may generate expressions such as $B +_{Int} 1$ in the example. These are problematic for the unification process (and ultimately, for the symbolic execution itself), because expressions cannot be unified - only variables can. An abstraction $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ of $\pi \wedge \phi$ is meant deal with this issue. For instance, $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ can be obtained from $\pi \wedge \phi$ by linearising the basic pattern π, replacing the non-data sub terms with variables from X, and then adding the equalities between variables in X and the corresponding subterms to $\widetilde{\phi}$ [2, 27]. This is exactly what happened in the above example. This issue is discussed further in Section 6.

We now define the notion of unifiability by a valuation ρ, both for patterns and for equivalence classes.

Definition 11 (Concrete and Symbolic ρ-Unifiability). Let $\rho : \mathit{Var} \to M$ be a valuation. Two formulas $\pi \wedge \phi, \pi' \wedge \phi'$ are said to be concretely ρ-unifiable if there is γ such that $(\gamma, \rho) \models (\pi \wedge \phi) \wedge (\pi' \wedge \phi')$. Two equivalence classes $[\pi \wedge \phi]_\sim, [\pi' \wedge \phi']_\sim$ are said to be symbolically ρ-unifiable if $\rho \models \psi$ for some $\psi \in \mathit{unif}([\pi \wedge \phi]_\sim, [\pi' \wedge \phi']_\sim)$.

Example 6. Let $\pi \wedge \phi \triangleq \langle\langle X \texttt{ = } 0\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\pi' \wedge \phi' \triangleq \langle\langle \texttt{n = } A\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. If we consider a valuation ρ, such that $\rho(X) = \texttt{n}$, $\rho(A) = 0$, and $\rho(M) = \rho(M')$, then $\pi \wedge \phi$ and $\pi' \wedge \phi'$ are ρ-unifiable. Next, let $(\exists I)\langle\langle X \texttt{ = } I\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I =_{\mathit{Int}} 0$ be an abstraction of $\pi \wedge \phi$ and $(\exists Y)\langle\langle Y \texttt{ = } A\rangle_{\mathsf{k}}\langle M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge Y =_{\mathit{Id}} \texttt{n}$ be an abstraction of $\pi' \wedge \phi'$. If $\sigma(X) = \sigma(Y) = U$, $\sigma(A) = \sigma(I) = V$, then $\rho \models (\exists I, Y, U, V)\phi^\sigma$ because there exists ρ', defined by $\rho'(I) = \rho'(V) = 0$, $\rho'(Y) = \rho'(U) = \texttt{n}$, $\rho'(X) = \rho(X)$, and $\rho'(A) = \rho(A)$ such that $\rho' \models \phi^\sigma \triangleq X = U \wedge Y = U \wedge I = 0 \wedge V = 0$. Also, $\psi \triangleq (\exists I, Y, U, V) \wedge I =_{\mathit{Int}} 0 \wedge Y =_{\mathit{Id}} \texttt{n} \wedge \phi^\sigma$ is a symbolic unifier of $[\pi \wedge \phi]_\sim$, $[\pi' \wedge \phi']_\sim$.

The following assumption relates concrete and symbolic unifiability. Since concrete unifiability concerns the concrete model M and symbolic unifiability concerns the chosen symbolic model $M^{\mathfrak{s}}$, the assumption restricts the way M is related to $M^{\mathfrak{s}}$. We will see in Section 6 actual examples of pairs of concrete and symbolic models satisfying this assumption.

Assumption 2 (Completeness). *For all concretely ρ-unifiable patterns $\pi \wedge \phi$ and $\pi' \wedge \phi'$, the classes $[\pi \wedge \phi]_\sim$ and $[\pi' \wedge \phi']_\sim$ are symbolically ρ-unifiable.*

5. Language Transformation

In this section we show how a new definition $((\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}), M^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ of a language $\mathcal{L}^{\mathfrak{s}}$ is automatically generated from a given a definition $(\Sigma, M, \mathcal{S})$ of a language $\mathcal{L}$. The new language $\mathcal{L}^{\mathfrak{s}}$ has the same syntax as $\mathcal{L}$, but its model $M^{\mathfrak{s}}$ is the symbolic model defined in the previous section, and its semantical rules $\mathcal{S}^{\mathfrak{s}}$ adapts the semantical rules $\mathcal{S}$ to deal with the new domain. Then, the symbolic execution of $\mathcal{L}$-programs is defined to be the concrete execution of the corresponding $\mathcal{L}^{\mathfrak{s}}$-programs. Building the definition of $\mathcal{L}^{\mathfrak{s}}$ amounts to:

1. extending the signature (Σ, Π) to a symbolic signature $(\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}})$;
2. extending the (Σ, Π)-model M to a $(\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}})$-model $M^{\mathfrak{s}}$;
3. turning the concrete rules $\mathcal{S}$ into symbolic rules $\mathcal{S}^{\mathfrak{s}}$.

We obtain the symbolic transition system $(M^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}, \Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}})$ by using Definitions 4, 5 for $\mathcal{L}^{\mathfrak{s}}$, just like the transition system $(M_{\mathit{Cfg}}, \Rightarrow^{M}_{\mathcal{S}})$ was defined for $\mathcal{L}$. We prove the Coverage and Precision results relating $\Rightarrow^{M}_{\mathcal{S}}$ to $\Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}}$.

5.1. Extending the Signature (Σ, Π) *to a Symbolic Signature* $(\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}})$

$\Sigma^{\mathfrak{s}}$ contains two new sorts: $\mathit{Cfg}^{\mathfrak{s}}$ and Bool. The operations of sort Bool include the usual propositional items $(\top, \wedge, \neg)$, the existential quantifier, as well as an operation $p : s_1 \ldots s_n \to \mathit{Bool}$ for each predicate $p \in \Pi_{s_1 \ldots, s_n}$. The unique operation of sort $\mathit{Cfg}^{\mathfrak{s}}$ is its constructor $\_ \wedge \_ : \mathit{Cfg} \times \mathit{Bool} \to \mathit{Cfg}^{\mathfrak{s}}$. The sort $\mathit{Cfg}^{\mathfrak{s}}$ is used to represent ML formulas $\pi \wedge \phi$ as terms. We naturally identify $\Sigma^{\mathfrak{s}}$-terms of the sort Bool with the corresponding FOL (Σ, Π)-formulas. $\Pi^{\mathfrak{s}}$ consist of one predicate `sat`, which takes one argument of sort Bool.

For the sake of presentation, for the IMP example we assume that the new sort Bool extends the existing one with the new operations.

5.2. Extending the Model M *to a Symbolic Model* $M^{\mathfrak{s}}$

The notation $[\![\_]\!]$ is extended to arbitrary terms: $[\![t]\!] \triangleq \{\rho(t) \mid \rho \text{ a valuation}\}$. Then, the definition of the equivalence $\sim$ is extended over arbitrary terms: $t \sim t'$ iff $[\![t]\!] = [\![t']\!]$.

$M^{\mathfrak{s}}$ interprets the elements of $\Sigma^{\mathfrak{s}}$ and $\Pi^{\mathfrak{s}}$ as follows:

1. sorts s in Σ are interpreted as $\sim$-equivalence classes of terms in $T_{\Sigma(\mathcal{D}),s}(\mathit{Var}^{\mathfrak{d}})$, where $\mathit{Var}^{\mathfrak{d}} \subsetneq \mathit{Var}$ is the (strict) subset of variables having data sorts, and the signature $\Sigma(\mathcal{D})$ is the extension of the signature Σ in which elements of the data domain $\mathcal{D}$ are declared as constants of the respective sorts;
2. the sort $\mathit{Cfg}^{\mathfrak{s}}$ is interpreted as $\sim$-equivalences classes of terms of sort $\mathit{Cfg}^{\mathfrak{s}}$, of the form $[\pi \wedge \phi]_\sim$, where $\pi \in T_{\Sigma(\mathcal{D}),\mathit{Cfg}}(\mathit{Var}^{\mathfrak{d}})$ and $\phi \in T_{\Sigma^{\mathfrak{s}}(\mathcal{D}),\mathit{Bool}}(\mathit{Var}^{\mathfrak{d}})$;
3. operations in $\Sigma^{\mathfrak{s}}$ are interpreted syntactically, i.e. as term constructors;
4. the (unique) predicate $\mathtt{sat} \in \Pi^{\mathfrak{s}}$ is interpreted as the theoretical satisfiability predicate for FOL formulas.

Example 7. If $I \in \mathit{Var}^{\mathfrak{d}}$, then $M^{\mathfrak{s}}$ interprets $I +_{\mathit{Int}} 3$ as the equivalence class $[I +_{\mathit{Int}} 3]_\sim$. For instance, $1 +_{\mathit{Int}} I +_{\mathit{Int}} 2 \in [I +_{\mathit{Int}} 3]_\sim$. If $B \in \mathit{Var}^{\mathfrak{d}}$, then the symbolic configuration

$$\langle \mathtt{if}\, B\, \mathtt{then\ x\ =\ 1;\ else\ x\ =\ 0;} \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 7 \rangle_{\mathsf{env}} \wedge B =_{\mathit{Bool}} \mathit{true}$$

is interpreted as $[\pi \wedge \phi]_\sim$, where

$$\pi \triangleq \langle \mathtt{if}\, B\, \mathtt{then\ x\ =\ 1;\ else\ x\ =\ 0;} \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 7 \rangle_{\mathsf{env}} \in T_{\Sigma(\mathcal{D}),\mathit{Cfg}}(\mathit{Var}^{\mathfrak{d}})$$

and

$$\phi \triangleq B =_{\mathit{Bool}} \mathit{true} \in T_{\Sigma^{\mathfrak{s}}(\mathcal{D}),\mathit{Bool}}(\mathit{Var}^{\mathfrak{d}}).$$

For instance,

$$\langle \mathtt{if}\, I <_{\mathit{Int}} 8\, \mathtt{then\ x\ =\ 1;\ else\ x\ =\ 0;} \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto I +_{\mathit{Int}} 1 \rangle_{\mathsf{env}} \wedge I =_{\mathit{Int}} 6 \in [\pi \wedge \phi]_\sim.$$

5.3. Turning the Concrete Rules $\mathcal{S}$ *into Symbolic Rules* $\mathcal{S}^{\mathfrak{s}}$

The set $\mathcal{S}^{\mathfrak{s}}$ consists of a rule

$$\widetilde{\pi}_1 \wedge \xi \Rightarrow \pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \xi$$

for each rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ and each abstraction $(\exists X)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ of $\pi_1 \wedge \phi_1$

(cf. Definition 10). Note that both the left-hand side and the right-hand side of the above rule are terms of sorts $Cfg^{\mathfrak{s}}$.

If one is interested only in *feasible executions*, then the rules in $\mathcal{S}^{\mathfrak{s}}$ will be of the form

$$(\widetilde{\pi}_1 \wedge \xi) \wedge sat(\phi_2 \wedge \widetilde{\phi}_1 \wedge \xi) \Rightarrow \pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \xi$$

Now, the left-hand side of the rule is an ML formula, where $sat(\phi_2 \wedge \widetilde{\phi}_1 \wedge \xi)$ is the condition.

We shall see later in this section that for practical cases it is enough to consider only one abstraction for each left-hand side of a rule in $\mathcal{S}$, and hence we obtain a one-to-one correspondence between $\mathcal{S}$ and $\mathcal{S}^{\mathfrak{s}}$.

Example 8. Let $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ be

$$\langle\langle \texttt{if}\, true\, \texttt{then}\, S_1 \texttt{else}\, S_2\rangle_{\mathsf{k}} \langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

and consider the abstraction $(\exists X)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ of $\pi_1 \wedge \phi_1$:

$$(\exists B)\langle\langle \texttt{if}\, B\, \texttt{then}\, S_1 \texttt{else}\, S_2\rangle_{\mathsf{k}} \langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (B =_{Bool} true).$$

Then, the corresponding rule in $\mathcal{S}^{\mathfrak{s}}$ is:

$$\langle\langle \texttt{if}\, B\, \texttt{then}\, S_1 \texttt{else}\, S_2\rangle_{\mathsf{k}} \langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \xi \Rightarrow \langle S_1\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}} \wedge (B =_{Bool} true) \wedge \xi$$

To obtain only feasible executions we use of the satisfiability predicate:

$$\langle\langle \texttt{if}\, B\, \texttt{then}\, S_1 \texttt{else}\, S_2\rangle_{\mathsf{k}} \langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \xi \wedge sat((B =_{Bool} true) \wedge \xi) \Rightarrow$$
$$\langle\langle S_1\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (B =_{Bool} true) \wedge \xi$$

5.4. Relating $\mathcal{L}$ and $\mathcal{L}^{\mathfrak{s}}$

In this section we prove the Coverage and Precision results that relate $\Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}}$ (defined by $\mathcal{L}^{\mathfrak{s}}$) with $\Rightarrow^{M}_{\mathcal{S}}$ (defined by $\mathcal{L}$) . Since we defined the symbolic transition based on unification, and the semantics of programming languages is based on rewriting, we have, in general, only a weaker coverage result. However, if the unification can be achieved by matching, as is the case in practically relevant cases discussed in Section 6, then we have the expected coverage result as stated in the introduction (Section 1): to every concrete execution there corresponds a feasible symbolic one.

Assumption 3. *Hereafter, whenever a rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ is applied to a pattern $\pi \wedge \phi$ with a unifier σ, we assume, w.l.o.g., that the variable names are chosen s.t. $var(\pi_1, \phi_1, \pi_2, \phi_2) \cap var(\pi, \phi) = \emptyset$, and the unifier σ is chosen s.t. $var(ran(\sigma)) \cap var(\pi_2, \phi_2) = \emptyset$. This is assumed for the symbolic rules as well.*

Before we prove the coverage and precision results we need the following lemmas. Remember (cf Definition 10) that, for a substitution σ, ϕ^{σ} denotes the FOL formula $\bigwedge_{x\in dom(\sigma)}(x = \sigma(x))$. Lemma 1 and Lemma 2 are auxiliary results which are used to prove that, under certain conditions, $\sigma(\pi \wedge \phi)$ and $\pi \wedge \phi \wedge \phi^{\sigma}$ have the same semantics (Lemma 3).

Lemma 1. *Let $\sigma : X \to T_\Sigma(Var)$. Then for all $t \in T_\Sigma(X)$ and all valuations $\rho : Var \to M$, if $\rho \models \phi^\sigma$ then $\rho(\sigma(t)) = \rho(t)$.*

PROOF. By structural induction on t.

If t is a variable then $t \in X$ and from $\rho \models \phi^\sigma (\triangleq \bigwedge_{x\in X} x = \sigma(x))$ we obtain $\rho(t) = \rho(\sigma(t))$, which proves the base step.

Assume that $t = f(t_1, \ldots, t_n)$ with $n \geq 0$ and $t_i \in T_\Sigma(X)$ for $i = 1, \ldots, n$. From the induction hypothesis we know that $\rho(t_i) = \rho(\sigma(t_i))$ for all $i = 1, \ldots, n$. Thus, $\rho(t) = f(\rho(t_1), \ldots, \rho(t_n)) = f(\rho(\sigma(t_1)), \ldots, \rho(\sigma(t_n))) = f((\rho \circ \sigma)(t_1), \ldots, (\rho \circ \sigma)(t_n)) = (\rho \circ \sigma)(f(t_1, \ldots, t_n)) = \rho(\sigma(f(t_1, \ldots, t_n)))$, which proves the inductive step and the lemma. $\square$

Lemma 2. *Let $\sigma : X \to M_\Sigma(Var)$ be such that $X \cap var(ran(\sigma)) = \emptyset$. Then for all* FOL *formulas ϕ and all valuations ρ, if $\rho \models \phi^\sigma$ then ($\rho \models \sigma(\phi)$ iff $\rho \models \phi$).*

PROOF. We proceed by structural induction on ϕ. The base case ($\phi = \top$) is trivial, so we only focus on the remaining cases:
1) ϕ is $p(t_1, \ldots, t_n)$ with $p \in \Pi$. Then

$$\begin{aligned}
\rho \models \sigma(\phi) &\iff \rho \models p(\sigma(t_1), \ldots, \sigma(t_n)) \\
&\iff (\rho(\sigma(t_1)), \ldots, \rho(\sigma(t_n))) \in M_p \\
&\iff (\rho(t_1), \ldots, \rho(t_n)) \in M_p \qquad \text{(by Lemma 1)} \\
&\iff \rho \models p(t_1, \ldots, t_n) \\
&\iff \rho \models \phi.
\end{aligned}$$

2) ϕ is $\neg\phi_1$. Then

$$\begin{aligned}
\rho \models \sigma(\phi) &\iff \rho \not\models \sigma(\phi_1) \\
&\iff \rho \not\models \phi_1 \qquad \text{(by the induction hypothesis)} \\
&\iff \rho \models \neg\phi_1 \\
&\iff \rho \models \phi.
\end{aligned}$$

3) ϕ is $\phi_1 \wedge \phi_2$. The proof is similar to that of case 2), using ($\rho \models \sigma(\phi_1)$ iff $\rho \models \phi_1$) and ($\rho \models \sigma(\phi_2)$ iff $\rho \models \phi_2$) as inductive hypotheses.
4) ϕ is $(\exists Y)\phi_1$. We may assume w.l.o.g. that $Y \cap (X \cup var(ran(\sigma))) = \emptyset$. We have $\sigma(\phi) = (\exists Y)\sigma(\phi_1)$. Then

$$\begin{aligned}
\rho \models \sigma(\phi) &\iff (\exists \rho')\rho' \models \sigma(\phi_1) \\
&\iff (\exists \rho')\rho' \models \phi_1 \qquad \text{(by the induction hypothesis)} \\
&\iff \rho \models \phi.
\end{aligned}$$

where ρ' satisfies $\rho'(x) = \rho(x)$ for all $x \notin Y$, which implies $\rho(\sigma(x)) = \rho(x)$ by the hypotheses of the lemma and $Y \cap (X \cup var(ran(\sigma))) = \emptyset$. $\square$

Lemma 3 and Lemma 4 are essential for proving our weak coverage result.

Lemma 3. *If $\sigma : X \to T_\Sigma(\mathit{Var})$ such that $X \cap \mathit{var}(\mathit{ran}(\sigma)) = \emptyset$ and $\mathit{var}(\pi \wedge \phi) \subseteq X$, then $\sigma(\pi \wedge \phi) \sim \pi \wedge \phi \wedge \phi^\sigma$.*

Proof. By Definition 6 of the $\sim$ relation, we have to prove that $[\![\sigma(\pi \wedge \phi)]\!] = [\![\pi \wedge \phi \wedge \phi^\sigma]\!]$. We distinguish two cases:

$\subseteq$. Assume $\gamma \in [\![\sigma(\pi \wedge \phi)]\!]$. Then there exists ρ such that $\gamma = \rho(\sigma(\pi))$ and $\rho \models \sigma(\pi)$. Let ρ' denote the valuation given by $\rho'(x) = \rho(\sigma(x))$, if $x \in X$, and $\rho'(x) = \rho(x)$, if $x \notin X$. We obviously have $\rho'(\pi) = \rho(\sigma(\pi)) = \gamma$. Since $X \cap \mathit{var}(\mathit{ran}(\sigma)) = \emptyset$ and $\mathit{var}(\pi \wedge \phi) \subseteq X$, we obtain $X \cap \mathit{var}(\sigma(\phi)) = \emptyset$, which implies $\rho \models \sigma(\phi)$ iff $\rho' \models \sigma(\phi)$. We obtain that $(\gamma, \rho') \models \pi \wedge \phi$. We also have $\rho'(x) = \rho(\sigma(x)) = \rho'(\sigma(x))$ for all $x \in X$ thanks again to $X \cap \mathit{var}(\mathit{ran}(\sigma)) = \emptyset$ and to the definition of ρ', which implies $\rho' \models \phi^\sigma$. We may conclude now that $\gamma \in [\![\pi \wedge \phi \wedge \phi^\sigma]\!]$. Since γ was arbitrarily chosen, we obtain $[\![\sigma(\pi \wedge \phi)]\!] \subseteq [\![\pi \wedge \phi \wedge \phi^\sigma]\!]$.

$\supseteq$. Assume $\gamma \in [\![\pi \wedge \phi \wedge \phi^\sigma]\!]$. Then there exists ρ such that $\gamma = \rho(\pi)$ and $\rho \models \phi$ and $\rho \models \phi^\sigma$. We obtain $\rho \models \sigma(\phi)$ by Lemma 2 and $\rho(\pi) = \rho(\sigma(\pi))$ by Lemma 1. Hence $\gamma \in [\![\sigma(\pi \wedge \phi)]\!]$. Since γ was arbitrarily chosen, we obtain $[\![\pi \wedge \phi \wedge \phi^\sigma]\!] \subseteq [\![\sigma(\pi \wedge \phi)]\!]$. □

Lemma 4. *Let $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ be an abstraction of the pattern $\pi \wedge \phi$. Then $\pi \wedge \phi \sim (\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$.*

Proof. We have to prove that $[\![\pi \wedge \phi]\!] = [\![\widetilde{\pi} \wedge \widetilde{\phi}]\!]$. Let σ be the substitution such that $\widetilde{\phi}$ is $\phi \wedge \phi^\sigma$. Then:

$$\begin{array}{ll}
\gamma \in [\![\pi \wedge \phi]\!] & \iff \\
(\exists \rho)(\gamma, \rho) \models \pi \wedge \phi & \iff \\
(\exists \rho')(\gamma, \rho') \models \widetilde{\pi} \wedge \widetilde{\phi} &
\end{array}$$

where $\rho'(x) = \rho(\sigma(x))$ for each $x \in X$ $(= \mathit{dom}(\sigma))$ and $\rho'(x) = \rho(x)$ otherwise. We obviously have $\rho'(\widetilde{\pi}) = \rho(\sigma(\widetilde{\pi})) = \rho(\pi) = \gamma$. Since $X \cap \mathit{var}(\mathit{ran}(\sigma)) = \emptyset$ we obtain $\rho'(\sigma(x)) = \rho(\sigma(x))$ for all $x \in X$, which implies $\rho' \models \phi^\sigma$. □

The next theorem says that every concrete transition $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ such that $\gamma \in [\![\pi \wedge \phi]\!]$ is simulated by a symbolic transition step. We call the result "weak" because in the symbolic transition step does not start exactly in $[\pi \wedge \phi]_\sim$, but in a subset of it.

Theorem 5 (One-step Weak Coverage). *If $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\pi \wedge \phi]\!]$, then there exists a* FOL *formula ϕ^σ and a pattern $\pi' \wedge \phi'$ such that $\gamma \in [\![\pi \wedge \phi \wedge \phi^\sigma]\!]$, $[\pi \wedge \phi \wedge \phi^\sigma]_\sim \Rightarrow_{\mathcal{S}^s}^{M^s} [\pi' \wedge \phi']_\sim$ and $\gamma' \in [\![\pi' \wedge \phi']\!]$.*

Proof. Assume that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. There exists $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ and a valuation ρ such that $(\gamma, \rho) \models \pi_1 \wedge \phi_1$ and $(\gamma', \rho) \models \pi_2 \wedge \phi_2$ by the definition of $\Rightarrow_{\mathcal{S}}$. From $\gamma \in [\![\pi \wedge \phi]\!]$ we obtain a valuation ρ' such that $(\gamma, \rho') \models \pi \wedge \phi$. Assumption 3 allows us to take $\rho = \rho'$ since the two valuations affect disjoint sets of variables. Thus, $(\gamma, \rho) \models \pi_1 \wedge \phi_1$ and $(\gamma, \rho) \models \pi \wedge \phi$, meaning that and $\pi \wedge \phi$ are concretely ρ-unifiable. By Assumption 2, there is $\psi \in \mathit{unif}([\pi_1 \wedge \phi_1]_\sim, [\pi \wedge \phi]_\sim)$ such that $\rho \models \psi$. We assume that $\psi \triangleq (\exists X_1 \cup X \cup \mathit{var}(\mathit{ran}(\sigma)))\widetilde{\phi}_1 \wedge \widetilde{\phi} \wedge \phi^\sigma$, where $M \models \pi_1 \wedge \phi_1 \leftrightarrow (\exists X_1)\widetilde{\pi}_1 \wedge \widetilde{\phi}_2$ and $M \models \pi \wedge \phi \leftrightarrow (\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$.

We have $\widetilde{\pi}_1 \wedge \xi \Rightarrow \pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \xi \in \mathcal{S}^{\mathsf{s}}$ by the definition of $\mathcal{S}^{\mathsf{s}}$. Let $\rho^{\mathsf{s}} : \mathit{Var} \to M^{\mathsf{s}}$ be the symbolic valuation such that $\rho^{\mathsf{s}}(x) = [\sigma(x)]_\sim$, for all $x \in \mathit{dom}(\sigma)$, and $\rho^{\mathsf{s}}(\xi) = \widetilde{\phi}$, and $\rho^{\mathsf{s}}(x) = [x]_\sim$ otherwise. We have

$$\begin{aligned}
\rho^{\mathsf{s}}(\widetilde{\pi}_1 \wedge \xi) &= [\sigma(\widetilde{\pi}_1) \wedge \widetilde{\phi}]_\sim && \text{(by the def. of } \rho^{\mathsf{s}}\text{)}\\
&= [\sigma(\widetilde{\pi}) \wedge \widetilde{\phi}]_\sim && (\sigma \text{ is a } \cong\text{-unifier})\\
&= [\widetilde{\pi} \wedge \widetilde{\phi} \wedge \phi^\sigma]_\sim && \text{(Lemma 3)}\\
&= [\pi \wedge \phi \wedge \phi^\sigma]_\sim && (\pi \wedge \phi \sim \widetilde{\pi} \wedge \widetilde{\phi} \text{ (using Lemma 4))}
\end{aligned}$$

and

$$\begin{aligned}
\rho^{\mathsf{s}}(\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \xi) &= [\sigma(\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1) \wedge \phi]_\sim && \text{(by the def. of } \rho^{\mathsf{s}}\text{)}\\
&= [\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi^\sigma \wedge \phi]_\sim && \text{(using Lemma 3)}
\end{aligned}$$

The above equalities imply $[\pi \wedge \phi \wedge \phi^\sigma]_\sim \Rightarrow^{M^{\mathsf{s}}}_{\mathcal{S}^{\mathsf{s}}} [\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi^\sigma \wedge \phi]_\sim$, so we can take $\pi' \triangleq \pi_2$ and $\phi' \triangleq \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi^\sigma \wedge \phi$. It remains to prove that $\gamma \in [\![\pi \wedge \phi \wedge \phi^\sigma]\!]$ and $\gamma' \in [\![\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi^\sigma \wedge \phi]\!]$. Since $\rho \models \psi$ and ψ is existentially quantified, it follows that there is ρ' such that $\rho' \models \widetilde{\phi}_1$, $\rho' \models \phi^\sigma$, and $\rho'(x) = \rho(x)$ for all $x \notin X_1 \cup X \cup \mathit{var}(\mathit{ran}(\sigma))$. We may choose X, X_1, and σ such that $\mathit{var}(\pi_2, \phi_2, \phi) \cap (X_1 \cup X \cup \mathit{var}(\mathit{ran}(\sigma))) = \emptyset$. From $(\gamma', \rho) \models \pi_2 \wedge \phi_2$, $\rho \models \phi$, and the definition of ρ' we obtain $(\gamma', \rho') \models \pi_2 \wedge \phi_2$ and $\rho' \models \phi$, which imply $(\gamma', \rho') \models \pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi^\sigma \wedge \phi$. Hence $\gamma' \in [\![\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi^\sigma \wedge \phi]\!]$. We obtain $(\gamma, \rho') \models \pi \wedge \phi \wedge \phi^\sigma$ in a similar way, which implies $\gamma \in [\![\pi \wedge \phi \wedge \phi^\sigma]\!]$, which finishes the proof. □

Example 9. Recall the semantic rule for the `and` operator, from the semantics of IMP (Figure 3). The rule can be written as below:

$$\begin{aligned}
&\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \triangleq\\
&\qquad \langle\langle \mathit{true}\ \mathtt{and}\ B \curvearrowright C\rangle_{\mathsf{k}} \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top \Rightarrow \langle\langle B \curvearrowright C\rangle_{\mathsf{k}} \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top^{\cdot}
\end{aligned}$$

Let $\pi \wedge \phi \triangleq \langle\langle B_1\ \mathtt{and}\ I <_{\mathit{Int}} 0 \curvearrowright C'\rangle_{\mathsf{k}} \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top$, and ρ be a valuation which satisfies $\rho(B_1) = \mathit{true}, \rho(I) = 2, \rho(B) = \mathit{false}$, $\rho(C) = \rho(C')$. Then $\pi_1 \wedge \phi_1$ and $\pi \wedge \phi$ are concretely ρ-unifiable and $\gamma \Rightarrow_{\mathcal{S}} \gamma'$, where $\gamma = \rho(\pi_1) = \rho(\pi)$ and $\gamma' = \rho(\pi_2)$. We assume that $\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ is $\pi_1 \wedge \phi_1$. Then the substitution σ given by $\sigma(B_1) = \mathit{true}, \sigma(B) = I <_{\mathit{Int}} 0, \sigma(I) = 2, \sigma(C) = \sigma(C') = C''$ is a unifier of $\widetilde{\pi}_1$ and π, where ϕ^σ is

$$B_1 =_{\mathit{Bool}} \mathit{true} \wedge B =_{\mathit{Bool}} I <_{\mathit{Int}} 0 \wedge I =_{\mathit{Int}} 2 \wedge C = C'' \wedge C' = C'',$$

and $\rho \models (\exists C'')\phi^\sigma$. Since ϕ, $\widetilde{\phi}$, ϕ_1, and ϕ_2 are all equal to $\top$, we have $\pi \wedge \phi^\sigma \Rightarrow^{M^{\mathsf{s}}}_{\mathcal{S}^{\mathsf{s}}}$

$\pi_2 \wedge \phi^\sigma$, i.e. (we replaced $C = C'' \wedge C' = C''$ by $C = C'$ for the sake of presentation):

$$[\langle\langle B_1 \texttt{ and } I <_{Int} 0 \curvearrowright C'\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge B_1 =_{Bool} true \wedge B =_{Bool} I <_{Int} 0 \wedge I =_{Int} 2 \wedge C = C']_\sim$$
$$\Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}}$$
$$[\langle\langle B \curvearrowright C\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge B_1 =_{Bool} true \wedge B =_{Bool} I{<_{Int}}0 \wedge I =_{Int} 2 \wedge C = C']_\sim$$

The "One-step Weak Coverage" property stated by Theorem 5 can be transformed into "One-step Coverage", i.e., the symbolic step covering the concrete one starts exactly from the initial formula (and not from a strengthening of it) if the abstractions of the involved patterns can be defined such that unification reduces to matching.

Corollary 6 (One-step Coverage). *Let $\pi \wedge \phi$ be a pattern such that for each rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ there exist abstractions $(\exists X_1)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ and $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ of $\pi_1 \wedge \phi_1$ and $\pi \wedge \phi$, respectively, such that $\sigma(\widetilde{\pi}_1) = \widetilde{\pi}$ for all $\cong$-unifiers σ of $\widetilde{\pi}_1$ and $\widetilde{\pi}$. Then, for all $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ with $\gamma \in [\![\pi \wedge \phi]\!]$ there exists a pattern $\pi' \wedge \phi'$ such that $[\pi \wedge \phi]_\sim \Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}} [\pi' \wedge \phi']_\sim$ and $\gamma' \in [\![\pi' \wedge \phi']\!]$.*

Proof. In the proof of Theorem 5, we use the set of equalities for $\rho^{\mathfrak{s}}(\widetilde{\pi}_1 \wedge \xi)$:

$$\begin{aligned} \rho^{\mathfrak{s}}(\widetilde{\pi}_1 \wedge \xi) &= [\sigma(\widetilde{\pi}_1) \wedge \widetilde{\phi}]_\sim && \text{(by the def. of } \rho^{\mathfrak{s}}) \\ &= [\widetilde{\pi} \wedge \widetilde{\phi}]_\sim && (\sigma \text{ is a } \cong\text{-matcher}) \\ &= [\pi \wedge \phi]_\sim && (\pi \wedge \phi \sim \widetilde{\pi} \wedge \widetilde{\phi}) \end{aligned}$$ □

In Section 6 we present the general conditions under which the hypotheses of Corollary 6 hold, and thus, full coverage holds. This is illustrated in Example 10.

Example 10. We assume that the rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ and the pattern $\pi \wedge \phi$ are those given in Example 9. If the abstraction $\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ of $\pi_1 \wedge \phi_1$ is $\langle\langle B' \texttt{ and } B \curvearrowright C\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge B' = true$, then the substitution σ given by $\sigma(B') = B_1, \sigma(B) = I <_{Int} 0, \sigma(I) = I', \sigma(C) = \sigma(C') = C''$ is a unifier of $\widetilde{\pi}_1$ and π, ϕ^σ is

$$B' =_{Bool} B_1 \wedge B =_{Bool} I <_{Int} 0 \wedge I =_{Int} I' \wedge C = C'' \wedge C' = C'',$$

and $\rho \models (\exists I', C'')\phi^\sigma$. Since σ just renames the variables of π, there exists σ' such that $\sigma'(\pi_1) = \pi$ and we have $\pi \Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}} \pi_2 \wedge \phi^{\sigma'}$, i.e.:

$$[\langle\langle B_1 \texttt{ and } I <_{Int} 0 \curvearrowright C'\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge C = C']_\sim$$
$$\Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}}$$
$$[\langle\langle B \curvearrowright C\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge B_1 =_{Bool} true \wedge B =_{Bool} I <_{Int} 0 \wedge C = C']_\sim.$$

Now we formulate the coverage result relating concrete and symbolic executions.

Corollary 7 (Coverage). *Under the hypothesis of Corollary 6, for each concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_i \Rightarrow_{\mathcal{S}} \cdots$ with $\gamma_0 \in [\![\pi_0 \wedge \phi_0]\!]$, there is a symbolic execution $[\pi_0 \wedge \phi_0]_\sim \Rightarrow_{\mathcal{S}^{\mathfrak{s}}}^{M^{\mathfrak{s}}} \cdots \Rightarrow_{\mathcal{S}^{\mathfrak{s}}}^{M^{\mathfrak{s}}} [\pi_i \wedge \phi_i]_\sim \Rightarrow_{\mathcal{S}^{\mathfrak{s}}}^{M^{\mathfrak{s}}} \cdots$ such that for all $i = 0, 1, \ldots$, $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$.*

Proof. By induction on i using Corollary 6. $\square$

Note that a similar coverage result based on Theorem 5 (instead of its Corollary 6) does not hold, because the symbolic steps given by this theorem cannot be "connected" into a symbolic execution.

The coverage property is one of the two properties expected from symbolic execution. It relates concrete steps to symbolic ones. The second one, stated by the next result, relates "feasible" symbolic steps to concrete ones.

Theorem 8 (One-step Precision). *If $[\pi \wedge \phi]_\sim \Rightarrow_{\mathcal{S}^{\mathfrak{s}}}^{M^{\mathfrak{s}}} [\pi' \wedge \phi']_\sim$ and $\gamma' \in [\![\pi' \wedge \phi']\!]$ then there exists a configuration γ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\pi \wedge \phi]\!]$.*

Proof. Let $\widetilde{\pi}_1 \wedge \xi \Rightarrow \pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \xi \in \mathcal{S}^{\mathfrak{s}}$ and $\rho^{\mathfrak{s}} : Var \to M^{\mathfrak{s}}$ be such that $M \models \pi_1 \wedge \phi_1 \leftrightarrow (\exists X_1)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ (i), $\rho^{\mathfrak{s}}(\widetilde{\pi}_1 \wedge \xi) = [\pi \wedge \phi]_\sim$ and $\rho^{\mathfrak{s}}(\widetilde{\pi}_2 \wedge \widetilde{\phi}_2 \wedge \widetilde{\phi}_1 \wedge \xi) = [\pi' \wedge \phi']_\sim$. Let $\sigma^{\mathfrak{s}}$ denote a substitution such that $\sigma^{\mathfrak{s}}(x) = \rho^{\mathfrak{s}}(x)$ for all variables $x \in var(\widetilde{\pi}_1.\widetilde{\phi}_1, \pi_2, \phi_2)$, and $\sigma^{\mathfrak{s}}(\xi) = \phi$. Then

$$\begin{aligned}
[\pi \wedge \phi]_\sim &= [\sigma^{\mathfrak{s}}(\widetilde{\pi}_1 \wedge \xi)]_\sim && \text{(the def. of } \sigma^{\mathfrak{s}}\text{)}\\
&= [\sigma^{\mathfrak{s}}(\widetilde{\pi}_1) \wedge \phi]_\sim && \text{(the def. of } \sigma^{\mathfrak{s}}\text{)}\\
&= [\widetilde{\pi}_1 \wedge \phi \wedge \phi^{\sigma^{\mathfrak{s}}}]_\sim && \text{(Lemma 3)} \qquad \text{(ii)}
\end{aligned}$$

and

$$\begin{aligned}
[\pi' \wedge \phi']_\sim &= [\sigma^{\mathfrak{s}}(\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \xi)]_\sim && \text{(the def. of } \sigma^{\mathfrak{s}}\text{)}\\
&= [\sigma^{\mathfrak{s}}(\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1) \wedge \phi]_\sim && \text{(the def. of } \sigma^{\mathfrak{s}}\text{)}\\
&= [\pi_2 \wedge \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi \wedge \phi^{\sigma^{\mathfrak{s}}}]_\sim. && \text{(Lemma 3)}
\end{aligned}$$

From $\gamma' \in [\![\pi' \wedge \phi']\!]$ and the above equalities we deduce that there are ρ' and ρ'' such that $\rho'(\pi') = \rho''(\pi_2) = \gamma'$, $\rho' \models \phi'$, and $\rho'' \models \phi_2 \wedge \widetilde{\phi}_1 \wedge \phi \wedge \phi^{\sigma^{\mathfrak{s}}}$. Since $var(\pi_2, \phi_2) \cap var(\pi', \phi') = \emptyset$ by Assumption 3, there exists ρ such that $\rho|_{var(\pi',\phi')} = \rho'$, $\rho|_{var(\pi_2,\phi_2)} = \rho''$, and hence $(\gamma', \rho) \models \pi_2 \wedge \phi_2 \wedge \pi' \wedge \phi'$. It follows that $\pi_2 \wedge \phi_2$ and $\pi' \wedge \phi'$ are ρ-unifiable and hence there is a symbolic unifier $\phi^\sigma \wedge \widetilde{\phi_2} \wedge \widetilde{\phi'}$ of $[\pi_2 \wedge \phi_2]_\sim$ and $[\pi' \wedge \phi']_\sim$ by Assumption 2, where $\sigma \in unif_{\cong}(\widetilde{\pi_2}, \widetilde{\pi}')$. We also have $\rho \models \phi^\sigma$ by Assumption 2 and hence $\gamma' \in [\![\pi' \wedge \phi' \wedge \phi^\sigma]\!]$.

Let $\gamma \triangleq \rho(\widetilde{\pi}_1)$. We also have $var(\widetilde{\pi}_1, \widetilde{\phi}_1) \cap var(\pi', \phi') = \emptyset$ by Assumption 3, and therefore we may assume w.l.o.g. that $\rho|_{var(\widetilde{\pi}_1,\widetilde{\phi}_1)} = \rho''$, and hence $(\gamma, \rho) \models \widetilde{\pi}_1 \wedge \widetilde{\phi}_1$. It follows that $(\gamma, \rho) \models (\exists X_1)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$, which implies $(\gamma, \rho) \models \pi_1 \wedge \phi_1$ thanks to to the equivalence (i). We have already obtained $(\gamma', \rho) \models \pi_2 \wedge \phi_2$, thus, we have the theorem's first conclusion: $\gamma \Rightarrow_{\mathcal{S}} \gamma'$.

There remains to prove the second conclusion: $\gamma \in [\![\pi \wedge \phi]\!]$. Since $\rho|_{var(\widetilde{\pi}_1,\widetilde{\phi}_1)} = \rho''$ it follows that $\gamma = \rho''(\widetilde{\pi}_1)$. We have $\rho'' \models \phi \wedge \phi^{\sigma^{\mathfrak{s}}}$ by the definition of ρ'', hence $\gamma \in [\![\widetilde{\pi}_1 \wedge \phi \wedge \phi^{\sigma^{\mathfrak{s}}}]\!]$ that is equal to $[\![\pi \wedge \phi]\!]$ by (ii), which proves the theorem's second conclusion. $\square$

We may now formulate the precision result which follows from Theorem 8:

Corollary 9 (Precision). *For every feasible symbolic execution* $[\pi_0 \wedge \phi_0]_\sim \Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}} \cdots \Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}} [\pi_i \wedge \phi_i]_\sim \Rightarrow^{M^{\mathfrak{s}}}_{\mathcal{S}^{\mathfrak{s}}} \cdots$ *there is a concrete execution* $\gamma_0 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_i \Rightarrow_{\mathcal{S}} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for all* $i = 0, 1, \ldots$.

Proof. By induction, using Theorem 8. $\square$

Remember that coverage and precision have practical consequences: they ensure that results of analyses performed by symbolically executing programs (reachability analysis, model checking,...) also hold for their concrete executions, which are, of course, the executions that one is actually interested in.

6. Instantiating Language Transformations

In this section we address some of the issues left open in Sections 4 and 5, in order to ensure a smooth transition between our theoretical language-transformation approach and its implementation given in Section 7. The issues are:

1. how to obtain a set of symbolic rules $\mathcal{S}^{\mathfrak{s}}$ with same cardinality as $\mathcal{S}$ (i.e., typically, $\mathcal{S}$ is finite, but the theoretical construction given in Section 5 will generate an infinite $\mathcal{S}^{\mathfrak{s}}$, which is impossible to use in practice);
2. how to satisfy Assumption 2 (required by the Theorem 5);
3. how to obtain unification by matching (required by Corollary 6).

We first show in Section 6.1 how to solve the first of the above-listed issues. Then, we present two instances of the current theoretical framework, and show how we address the two remaining issues from our list:

- the first instance, presented in Section 6.2, coincides with that presented in the earlier paper [2];
- the second instance, presented in Section 6.3, is an extension of the previous one. It allows one to use axiomatically-defined structures (sets, bags, lists, ...), with axioms such as associativity, commutativity, unity and combinations thereof. Such structures are intensively used in actual language definitions, such as those defined in the $\mathbb{K}$ framework.

In the rest of this section we consider given a $(\Sigma^{\mathfrak{d}}, \Phi^{\mathfrak{d}})$-model $\mathcal{D}$, which interprets the data sorts, operations, and predicates.

6.1. Obtaining a set of symbolic of same cardinality as the set of concrete rules

The set of symbolic rules $\mathcal{S}^{\mathfrak{s}}$, defined in Section 5.3 is typically infinite, because of the infinitely many abstractions $(\exists X)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ of the left-hand sides $\pi_1 \wedge \phi_1$ of rules in $\mathcal{S}$. The algorithm presented below, which we call LDA (for Linearisation and Data Abstraction) produces a unique abstraction for any pattern (and thus, a one-to-one correspondence between $\mathcal{S}$ and $\mathcal{S}^{\mathfrak{s}}$).

We present the LDA algorithm and illustrate it by examples.

1. Linearising basic patterns. Recall that a term is linear if any variable occurs at most once in its left-hand side. A pattern $\pi \wedge \phi$ is linear if π is linear. A nonlinear pattern can always be turned into an equivalent linear one, by renaming the variables occurring several times in π and adding the equalities between the renamed variables and the original ones to the condition ϕ.

For example,

$$\langle\langle X\rangle_{\mathsf{k}}\langle X \mapsto I\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top$$

is transformed into

$$\langle\langle X\rangle_{\mathsf{k}}\langle X' \mapsto I\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top \wedge X = X',$$

where we assumed that $X \mapsto I$ is not a data term.

2. Data Abstraction. Let $Dpos(t)$ be the set of positions p[2] of the term t such that $t|_p$ is a maximal subterm of a data sort. The next step of our pattern transformation consists in replacing all the maximal data subterms $\pi|_p$ of π by fresh variables x_p and adding the equalities between the fresh variables and the corresponding subterms to the condition ϕ. Formally, $\pi \wedge \phi$ is transformed into $\pi[x_p/\pi|_p]_{p\in Dpos(\pi)} \wedge \phi \wedge \bigwedge_{p\in Dpos(\pi)}(x_p = \pi|_p))$. For instance, the pattern

$$\langle\langle \texttt{if}\ \mathit{true}\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top$$

ts transformed into

$$\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle_{\mathsf{k}}\langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top \wedge B = \mathit{true}.$$

The above transformations describe an algorithm that builds an unique (up to a renaming of the new added variables) abstraction $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ for a given pattern $\pi \wedge \phi$. Moreover, $\widetilde{\phi}$ is of the form $\phi \wedge \phi^{\sigma}$ for a substitution σ that sends each fresh variable x' into either a variable $x \in \mathit{var}(\pi)$ (due to the linearisation) or a data subterm π_p (due to the data abstraction).

We hereafter assume that all rules in $\mathcal{S}$ are transformed such that their left-hand sides are abstractions computed by LDA.

Example 11. The last rule from the original IMP semantics (Fig. 3) could have been written as a nonlinear rule:

$$\langle\langle X\ \cdots\rangle_{\mathsf{k}}\langle X \mapsto I\ \cdots\rangle_{\mathsf{env}}\ \cdots\rangle_{\mathsf{cfg}} \qquad\qquad \Rightarrow \langle\langle I\ \cdots\rangle_{\mathsf{k}}\langle X \mapsto I\ \cdots\rangle_{\mathsf{env}}\ \cdots\rangle_{\mathsf{cfg}}$$

in which case it would have been transformed into

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

The following rule for *if* from the IMP semantics:

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

[2]For the notion of position in a term and other rewriting-related notions, see, e.g., [5]. $t|_p$ denotes the subterm of t at position p.

is transformed into:

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

We still have to prove that the instances of our theoretical framework (presented in Sections 6.2 and 6.3 below) still satisfy coverage results (Theorem 5 and Corollary 6) when abstractions of patterns are computed by LDA instead of generating all possible abstractions. The proofs are given in the subsections below, since they depend on other results, which are specific to the particular instances. We note that precision (Theorem 8) is not affected by computing abstractions with LDA. Indeed, if all (feasible) symbolic executions generated by a larger set $\mathcal{S}^{\mathfrak{s}}$ have corresponding concrete ones, then this also holds for a subset of those symbolic executions, generated by a smaller set $\mathcal{S}^{\mathfrak{s}}$.

The two instances differ with respect to their definition of the model M, which is a parameter of any language definition $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), M, \mathcal{S})$. Remember from Section 3 that we asummed a subsignature $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ of (Σ, Π), and a $(\Sigma^{\mathfrak{d}}, \Phi^{\mathfrak{d}})$-model $\mathcal{D}$, which interprets the data sorts, operations, and predicates.

6.2. First instance: M consists of ground terms

For the first instance, the model M is defined as follows:

- for each item (sort, function/predicate symbol) $o \in (\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$, $M_o = \mathcal{D}_o$;
- for each sort s in $\Sigma \setminus \Sigma^{\mathfrak{d}}$, M_s is the set of ground $(\Sigma \setminus \Sigma^{\mathfrak{d}})(\mathcal{D})$-terms;
- for each function symbol f in $\Sigma \setminus \Sigma^{\mathfrak{d}}$, M_f is the term constructor f such that for all $(t_1, \ldots, t_n)$, $M_f(t_1, \ldots, t_n) = f(M_f(t_1), \ldots, M_f(t_n))$ (recall that the result sort of f does not belong to $\Sigma^{\mathfrak{d}}$ by the hypotheses).

Since M is uniquely determined by $\mathcal{D}$, we also denote M by $\mathcal{D}\rceil^{(\Sigma,\Pi)}$ in the sequel.

Example 12. The ground term $\langle\langle \mathtt{y} = \mathtt{x} + 3;\rangle_{\mathsf{k}}\langle \mathtt{x} \mapsto 5\ \mathtt{y} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a an element in M_{Cfg}, while the ground term $\langle\langle \mathtt{y} = \mathtt{x} + 3\rangle_{\mathsf{k}}\langle \mathtt{x} \mapsto 1 +_{\mathit{Int}} 2\ \mathtt{y} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ does not belong to M_{Cfg} because it includes the non-constant data term $1 +_{\mathit{Int}} 2$.

Even if the elements of M are terms, the valuations $\rho : \mathit{Var} \to M$ are not, in general, substitutions because they involves expressions on data (e.g., $1 +_{\mathit{Int}} 2$ in Example 12). Therefore the valuations do not define unifiers directly.

The following result shows that Assumption 2 from Section 4 holds for this case. Moreover it ensures that Theorem 5 holds when computing abstractions with the LDA algorithm:

Lemma 10 (Concrete Unifiability Implies Symbolic Unifiability). *Let $\pi_1 \wedge \phi_1$ and $\pi \wedge \phi$ be two patterns such that $var(\pi_1 \wedge \phi_1) \cap var(\pi \wedge \phi) = \emptyset$. If $\pi_1 \wedge \phi_1$ and $\pi \wedge \phi$ are concretely ρ-unifiable then the classes $[\pi_1 \wedge \phi_1]_\sim$ and $[\pi \wedge \phi]_\sim$ are symbolically ρ-unifiable, with a unifier $\psi = (\exists X \cup X' \cup var(ran(\sigma)))\phi^\sigma \wedge \widetilde{\phi_1} \wedge \widetilde{\phi}$, where $\sigma \in unif(\widetilde{\pi}_1, \widetilde{\pi})$, $\widetilde{\pi}_1 \wedge \widetilde{\phi_1}$, and $\widetilde{\pi} \wedge \widetilde{\phi}$ are computed using the LDA algorithm.*

PROOF. Let ρ be a valuation such that $\pi_1 \wedge \phi_1$ and $\pi \wedge \phi$ are concretely ρ-unifiable. There exists γ such that $(\gamma, \rho) \models \pi_1 \wedge \phi_1 \wedge \pi \wedge \phi$. Let $(\exists X_1)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ and $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ the abstractions of $\pi_1 \wedge \phi_1$ and $\pi \wedge \phi$, respectively, computed with the algorithm LDA. There exists $\widetilde{\rho}$ such that $(\gamma, \widetilde{\rho}) \models \widetilde{\pi}_1 \wedge \widetilde{\phi}_1 \wedge \widetilde{\pi} \wedge \widetilde{\phi}$ and $\widetilde{\rho}(x) = \rho(x)$, for all $x \notin X \cup X_1$. Let σ be the substitution defined by $\sigma(x) = \widetilde{\rho}(x)$ for $x \in var(\widetilde{\pi}_1, \widetilde{\pi})$. Since $\sigma(x) \in M$ and by the particular form of $\widetilde{\pi}_1$ and $\widetilde{\pi}$ we have $\sigma(\widetilde{\pi}_1) = \sigma(\widetilde{\pi})$, a ground term in M. Hence σ is a unifier of $\widetilde{\pi}_1$ and $\widetilde{\pi}$ and we have $\rho \models (\exists X_1 \cup X)\phi^\sigma$ because $\widetilde{\rho} \models \phi^\sigma$. Indeed, because $\sigma(x)$ is a ground term, $\widetilde{\rho}(\sigma(x)) = \sigma(x) = \widetilde{\rho}(x)$ because $var(ran(\sigma)) \cap (X \cup X_1) = \emptyset$ by Assumption 3. Hence $\psi \triangleq (\exists X \cup X_1)\widetilde{\phi}_1 \wedge \widetilde{\phi} \wedge \phi^\sigma$ is a symbolic ρ-unifier of $[\pi_1 \wedge \phi_1]_\sim$ and $[\pi \wedge \phi]_\sim$ (note that $var(ran(\sigma)) = \emptyset$ because σ is a ground substitution). $\square$

Example 13. Consider

$$\pi_1 \wedge \phi_1 \triangleq \langle\langle X \texttt{ + } 0\rangle_{\mathsf{k}}\langle .Map\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top$$

and

$$\pi \wedge \phi \triangleq \langle\langle 2 \texttt{ + } b +_{Int} 1\rangle_{\mathsf{k}}\langle .Map\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \top.$$

(N.B.: the operation + is part of the IMP syntax whereas $+_{Int}$ is addition in the integer data domain). First, note that $\pi_1 \wedge \phi_1$ and $\pi \wedge \phi$ are concretely ρ-unifiable, with ρ satisfying $\rho(X) = 2$ and $\rho(b) = -1$. Second, there is $\gamma \triangleq \langle\langle 2 \texttt{ + } 0\rangle_{\mathsf{k}}\langle .Map\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ such that $(\gamma, \rho) \models \pi_1 \wedge \phi_1$ and $(\gamma, \rho) \models \pi \wedge \phi$, since

$$\rho(2 \texttt{ + } b +_{Int} 1) = 2 \texttt{ + } \mathcal{D}_{+_{Int}}(\rho(b), 1) = 2 \texttt{ + } \mathcal{D}_{+_{Int}}(-1, 1) = 2 \texttt{ + } 0.$$

The abstractions of the two patterns computed as above are:

$$(\exists X_1)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1 \triangleq (\exists I)\langle\langle X\texttt{+}I\rangle_{\mathsf{k}}\langle .Map\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I =_{Int} 0$$

and

$$(\exists X)\widetilde{\pi} \wedge \widetilde{\phi} \triangleq (\exists A, B)\langle\langle A\texttt{+}B\rangle_{\mathsf{k}}\langle .Map\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A =_{Int} 2 \wedge B =_{Int} b +_{Int} 1.$$

The valuation $\widetilde{\rho}$ is given by $\widetilde{\rho}(X) = \rho(X) = 2$, $\widetilde{\rho}(b) = \rho(b) = -1$, $\widetilde{\rho}(I) = \rho(0) = 0$, $\widetilde{\rho}(A) = \rho(2) = 2$, $\widetilde{\rho}(B) = \rho(b +_{Int} 1) = 0$.

Let σ be the substitution given by $\sigma(x) = \widetilde{\rho}(x)$, for all $x \in var(\widetilde{\pi}_1, \widetilde{\pi})$, i.e., $\sigma(X) = 2, \sigma(I) = 0, \sigma(A) = 2, \sigma(B) = 0$. Then σ is a syntactic unifier of $\widetilde{\pi}_1$ and $\widetilde{\pi}$: $\sigma(X \texttt{ + } I) = \sigma(A \texttt{ + } B) = 2 \texttt{ + } 0$. We obviously have $\rho \models (\exists I, A, B)\phi^\sigma$, where ϕ^σ is $X =_{Int} 2 \wedge I =_{Int} 0 \wedge A =_{Int} 2 \wedge B =_{Int} 0$.

In the proof of Theorem 5 we consider a symbolic unifier ψ whose existence is given by Assumption 2. Lemma 10 is a stronger version of Assumption 2: the unifier ψ is computed using the LDA algorithm. In order to prove that Theorem 5 still holds it is enough to choose, in its proof, $\psi = (\exists X \cup X' \cup var(ran(\sigma)))\phi^\sigma \wedge \widetilde{\phi}_1 \wedge \widetilde{\phi}$, where $\sigma \in unif(\widetilde{\pi}_1, \widetilde{\pi})$, $\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$, and $\widetilde{\pi} \wedge \widetilde{\phi}$ are computed using the LDA algorithm.

We still have to show that Corollary 6 holds for this instance. For this, we show that its hypothesis is satisfied, i.e., we have to reduce unification in M to matching. This result is proved in a more general setting by Lemma 14 in the subsection below (we take the set A of axioms to be the empty set).

This concludes the description of this instance of our language-transformation approach. This instance is isomorphic with the earlier approach [2].

6.3. Second instance: M consists of equivalence classes of terms modulo axioms

Language definitions in the $\mathbb{K}$ framework often use structures such as bags and sets in configurations. In order to symbolically execute programs in such languages, one needs to solve constraints involving bags, sets, etc. One possibility is to consider those structures as *data*, and to deal with the constraints involving them by SMT solving. But this is a poor solution in practice because SMT solvers have limited support for constraints over bags and sets. Hence, some structures involved in language definitions must be defined axiomatically.

Thus, we assume given a set A of structural axioms (e.g., associativity, and/or commutativity, and/or identity)) for certain non-data functional symbols and for which there exists a *matching algorithm* modulo A that produces a finite number of A-matching solutions, whenever such solutions exist. The model M is $\mathcal{D}\rceil^{(\Sigma,\Pi)}/A$. Let $[t]_A \in M$ denote the A-equivalence class including t.

The following lemma generalises Lemma 10 from Section 6.2.

Lemma 11 (Concrete Unifiability Implies Symbolic Unifiability). *Let $\varphi_1 \triangleq \pi_1 \wedge \phi_1$ and $\varphi \triangleq \pi \wedge \phi$ be two patterns such that $var(\pi_1 \wedge \phi_1) \cap var(\pi \wedge \phi) = \emptyset$. If φ_1 and φ are concretely ρ-unifiable (N.B. in M) then there exists a symbolic ρ-unifier ψ of $[\varphi_1]_\sim$ and $[\varphi]_\sim$, where ψ is computed using the algorithm LDA.*

Proof. We use the same notations as in the proof of Lemma 10. Note that there are γ such that $(\gamma,\rho) \models \pi_1 \wedge \phi_1 \wedge \pi \wedge \phi$ and $\widetilde{\rho}$ such that $(\gamma,\widetilde{\rho}) \models \widetilde{\pi}_1 \wedge \widetilde{\phi}_1 \wedge \widetilde{\pi} \wedge \widetilde{\phi}$ and $\widetilde{\rho}(x) = \rho(x)$, for all $x \notin X \cup X_1$ (where $(\exists X_1)\widetilde{\pi}_1 \wedge \widetilde{\phi}_1$ and $(\exists X)\widetilde{\pi} \wedge \widetilde{\phi}$ are the abstractions of φ_1 and φ computed using LDA).

Since the elements in M are equivalence classes, the substitution σ is defined by $\sigma(x) \in \widetilde{\rho}(x)$ for $x \in var(\widetilde{\pi}_1, \widetilde{\pi})$. We have $[\sigma(\widetilde{\pi}_1)]_A = \widetilde{\rho}(\widetilde{\pi}_1) = \widetilde{\rho}(\widetilde{\pi}) = [\sigma(\widetilde{\pi})]_A$, which implies $\sigma(\widetilde{\pi}_1) =_A \sigma(\widetilde{\pi})$. Hence σ is a $=_A$-unifier of $\widetilde{\pi}_1$ and $\widetilde{\pi}$. We prove now that $\rho \models (\exists X_1 \cup X)(\phi^\sigma \wedge \widetilde{\phi}_1 \wedge \widetilde{\phi})$. Since we already have $\widetilde{\rho} \models \widetilde{\phi}_1$ and $\widetilde{\rho} \models \widetilde{\phi}$, it is enough to prove that $\widetilde{\rho} \models \phi^\sigma$. We have $\widetilde{\rho}(\sigma(x)) = [\sigma(x)]_A = \widetilde{\rho}(x)$ because $var(ran(\sigma)) \cap (X \cup X_1) = \emptyset$ by Assumption 3. Hence $\psi \triangleq (\exists X \cup X_1)\widetilde{\phi}_1 \wedge \widetilde{\phi} \wedge \phi^\sigma$ is a symbolic ρ-unifier of $[\pi_1 \wedge \phi_1]_\sim$ and $[\pi \wedge \phi]_\sim$. □

Thus, the Theorem 5 holds with the same argument employed in Section 6.2.

The last step is to prove that the conditions of Corollary 6 hold; that is, to reduce unification in M to matching. We hereafter assume that the axioms A are *linear*, *regular*, and *data collapse-free*; these requirements are usual for rewrite theories and all above mentioned axioms satisfy them. A term t is *linear* iff any variable occurs in t at most once, an $u = v$ is *regular* iff $var(u) = var(v)$ and it is *linear* if both sides u and v are linear. An axiom $u = v$ is *data collapse-free* iff it does not collapse a non-data term into a data term; formally, for any substitution σ neither $\sigma(u)$ nor $\sigma(v)$ is a variable of data sort. The next sequence of lemmas leads to the proof that unification in M reduces to matching.

In the following lemma we assume that $\widetilde{t}$ is obtained from the term t using the algorithm LDA, but forgetting the condition part.

Lemma 12. *Let $t, t' \in T_\Sigma$. If $t =_A t'$ then $\widetilde{t} =_A \widetilde{t'}$.*

Proof. The congruence $=_A$ is the smallest equivalence relation that include (i) $\sigma(u) =_A \sigma(v)$ for each axiom $u = v$ in A and ground substitution σ, and (ii) $t_0[t_1]_p =_A t_0[t_2]_p$ whenever $t_1 =_A t_2$.

We proceed by well-founded induction. We distinguish two cases:
1. $t = \sigma(u)$ and $t' = \sigma(v)$ for certain $u = v$ in A. Let $\widetilde{\sigma}$ the substitution defined by $\widetilde{\sigma}(x) = \widetilde{\sigma(x)}$ for all $x \in var(u) = var(v)$ (we have $var(u) = var(v)$ because $u = v$ is regular). Since u and v include only non-data functional symbols ($u = v$ is regular and data collapse-free) and are linear, we obtain $\widetilde{\sigma(u)} = \widetilde{\sigma}(u)$ and $\widetilde{\sigma(v)} = \widetilde{\sigma}(v)$. From (i) we have $\widetilde{\sigma}(u) =_A \widetilde{\sigma}(v)$, which implies $\widetilde{\sigma(u)} =_A \widetilde{\sigma(v)}$.
2. $t = t_0[t_1]_p$ and $t' = t_0[t_2]_p$ for certain t_0, $t_1 =_A t_2$, and position p in t_0. We have $\widetilde{t_1} =_A \widetilde{t_2}$ by the induction hypothesis. Since $\widetilde{t} = \widetilde{t_0}[\widetilde{t_1}]_p$ and $\widetilde{t'} = \widetilde{t_0}[\widetilde{t_2}]_p$ we obtain $\widetilde{t} =_A \widetilde{t'}$ using the definition of $=_A$. □

Lemma 13. *Let t be a linear term whose all data subterms are variables and these are the only variables occurring in t (hence $\widetilde{t} = t$). If σ is a ground substitution then there exists a variable renaming η such that $\eta(\widetilde{\sigma(t)}) = t$.*

Proof. The set $dom(\sigma)$ includes only data variables by the hypotheses of the lemma. We proceed by structural induction on t.
1. t is a variable. Then t has a data sort and $\widetilde{\sigma(t)}$ is a variable as well. Thus, we consider η such that $\eta(\widetilde{\sigma(t)}) = t$.
2. $t = f(t_1, \ldots, t_n)$, $n \geq 0$. Then f is a non-data functional symbol $\widetilde{\sigma(t)} = f(\widetilde{\sigma(t_1)}, \ldots \widetilde{\sigma(t_n)})$. From the inductive hypothesis, there are the variable renamings $\eta_1, \ldots, \eta_n$ such that $\eta_i(\widetilde{\sigma(t_i)}) = t_i$, $i = 1, \ldots, n$. Since t is linear, it follows that the substitution η, given by $\eta(x) = \eta_i(x)$ iff $x \in var(\widetilde{\sigma(t_i)})$ for all $1 \leq i \leq n$, is a variable renaming as well. We have $\eta(\widetilde{\sigma(t)}) = f(\eta_1(\widetilde{\sigma(t_1)}), \ldots \eta_n(\widetilde{\sigma(t_n)})) = f(t_1, \ldots, t_n) = t$.

The proof by induction is finished and the lemma is proved. □

Finally, we show that, under certain conditions, the unification (in M) reduces to matching:

Lemma 14 (Unification by Matching). *Let π_1 be a linear basic pattern whose data subterms are all variables (and hence $\widetilde{\pi}_1 = \pi_1$) and π a linear basic pattern whose all variables are of data sort and whose all data subterms are variables (and hence $\widetilde{\pi} = \pi$).*

We further assume that $var(\pi_1) \cap var(\pi) = \emptyset$. If σ is a $=_A$-unifier, i.e. σ is a ground substitution satisfying $\sigma(\pi_1) =_A \sigma(\pi)$, then there exists a variable renaming η such that $\eta(\widetilde{\sigma(\pi_1)}) =_A \pi$.

Proof. We have $\widetilde{\sigma(\pi_1)} =_A \widetilde{\sigma(\pi)}$ by Lemma 12 and $\eta(\widetilde{\sigma(\pi)}) = \pi$ by Lemma 13. □

Since every $=_A$-unifier σ is a $=_A$-matching, it follows that Corollary 6 (Coverage) holds. If there exists a *matching algorithm* modulo A that produces a finite number of A-matching solutions, we obtain that a left-hand side of a rule $\pi_1 \wedge \phi_1$ and symbolic configuration $\pi \wedge \phi$ are ρ-unifiable iff there exists a matching solution σ such that $\rho(x) = [\sigma(x)]_A$ and $\widetilde{\sigma(\pi_1)} =_A \widetilde{\pi}$.

There are several similarities between our approach in that given in [27]: their builtin theory is equivalent to the specification of data, constrained terms are patterns $\pi \wedge \phi$, and the abstraction of built-ins for a configuration term π is the same with our $\widetilde{t}$. According to the paragraph preceding Lemma 4 (Matching Lemma) in [27], by A-matching a configuration term π, including only data variables, against a left-hand side $\widetilde{\pi}_1$ of a rule in $\mathcal{S}$ provides a complete unifiability algorithm for ground A-unification of π and $\widetilde{\pi}_1$ (the claim was adapted to our notation). More technically, the Matching Lemma in [27] claims that if π and $\widetilde{\pi}_1$ are ground A-unifiable, the there is a matching substitution σ such that $\sigma(\widetilde{\pi}_1) =_A \pi$ (note the equality modulo A). Since ground A-unifiability is the same with concrete unifiability in the model M, we show below that we may take $unif_{\cong}(\widetilde{\pi}_1, \pi)$ to be the set of substitutions given by the matching algorithm. However, we cannot apply directly the Matching Lemma in [27] because it does not establish a direct relationship between unifiers and matchers.

This concludes the presentation of the practically relevant instances of our language-transformation approach.

7. Implementation

In this section we present a prototype tool implementing our approach. In Section 7.1 we briefly present our tool and its integration within the $\mathbb{K}$ framework (version 3.4). In Section 7.2 we illustrate the most significant features of the tool by the means of use cases involving nontrivial languages and programs.

7.1. Symbolic Execution within the $\mathbb{K}$ Framework

Our tool is part of $\mathbb{K}$ [29, 35], a semantic framework for defining operational semantics of programming languages. Specifically, our implementation is part of version 3.4 of the $\mathbb{K}$ compiler [3]. A $\mathbb{K}$ definition of a language, say, $\mathcal{L}$, is compiled into a Maude rewrite theory. Then, the $\mathbb{K}$ runner executes programs in $\mathcal{L}$ by applying the resulting rewrite rules to configurations containing programs.

Our tool follows the same process. The main difference is that our new $\mathbb{K}$ compiler includes the transformations presented in Section 6.2. The effect is that the compiled definition corresponds to the symbolic semantics of $\mathcal{L}$ instead of its concrete semantics. We note that the symbolic semantics can execute programs with concrete inputs as well. In this case it behaves like the concrete semantics. The theoretical relationships between $\mathbb{K}$ language definitions and their compilation to Maude, and between symbolic transformations of $\mathbb{K}$ definitions and their Maude encodings are investigated in [4].

The tool provides symbolic support for some of the most standard $\mathbb{K}$ data types: Booleans, integers, as well as arrays whose size, indices, and contents can be symbolic. The symbolic semantics is in general nondeterministic: when presented with symbolic inputs, a program can take several paths. Therefore the $\mathbb{K}$ runner can be called with several options: it can execute one nondeterministically chosen path, or all possible paths, up to a given depth; it can also be run in a step-by-step manner. During the execution, the path conditions (which are computed by the symbolic semantics) are checked for satisfiability using ad-hoc simplification rules and possibly, calls to the Z3 SMT solver[11].

In practice, path conditions may contain constraints over structures such as bags or sets. Unfortunately, SMT solvers have poor support for constraints over such structures, and thus, we always apply axioms of the symbolic domains to simplify the formulas before sending it to the solver. Then, for efficiency reasons, the SMT solver is called only if the rules adds non-trivial formula to path conditions, which cannot be simplified to *true* or *false* by the axioms of the symbolic domains.

7.2. Use cases

We show three use cases for our tool: the first one illustrates the execution and LTL model checking for IMP programs extended with I/O instructions, the second one demonstrates the use of symbolic arrays in the SIMPLE language – an extension of IMP with functions, arrays, threads and several other features, and the third one shows symbolic execution in an object-oriented language called KOOL [17]. The SIMPLE and KOOL languages have existed almost as long as the $\mathbb{K}$ framework and have intensively been used for teaching programming language concepts. All the examples presented below can be tested using the online interface at `http://fmse.info.uaic.ro/tools/Symbolic/`.

7.2.1. IMP *with I/O operations*

We first enrich the IMP language (Figure 1) with `read` and `print` operations. This enables the execution of IMP programs with symbolic input data. We then compile the resulting definition by calling the $\mathbb{K}$ compiler with an option

[3]A virtual machine running $\mathbb{K}$ (version 3.4) can be downloaded from `http://www.kframework.org/imgs/releases/kvm-3.4.zip`

```
int n, s;
n = read();
s = 0;
while (n > 0) {
  s = s + n;
  n = n - 1;
}
print("Sum = ", s, "\n");
```

Figure 4: `sum.imp`

```
int k, a, x;
a = read();
x = a;
while (x > 1) {
  x = x / 2;
  k = k + 1;
  L : {}
}
```

Figure 5: `log.imp`

telling it to generate the symbolic semantics of the language by applying the transformations described in Section 5.3.

Programs such as `sum.imp` shown in Figure 4 can now be run with the $\mathbb{K}$ runner in the following ways:

1. with symbolic or with concrete inputs;
2. on one arbitrary execution path, or on all paths up to a given bound;
3. in a step-wise manner, or by letting the program completely execute a given number of paths.

For example, by running `sum.imp` with a symbolic input n (here and thereafter we use mathematical font for symbolic values) and requiring at most five completed executions, the $\mathbb{K}$ runner outputs the five resulting, final configurations, one of which is shown below, in a syntax slightly simplified for readability:

`<k> .  </k>`

`<path-condition>` $n > 0 \wedge (n-1 > 0) \wedge \neg((n-1)-1 > 0)$ `</path-condition>`

`<state>`

`n |->` $(n-1)-1$

`s |->` $n+(n-1)$

`</state>`

The program is finished since the `k` cell has no code left to execute. The path condition actually means $n = 2$, and in this case the sum `s` equals $n + (n-1) = 2 + 1$, as shown by the `state` cell. The other four final configurations, not shown here, compute the sums of numbers up to 1, 3, 4, and 5, respectively. Users can run the program in a step-wise manner in order to see intermediary configurations in additional to final ones. During this process they can interact with the runner, e.g., by choosing one execution branch of the program among several, feeding the program with inputs, or letting the program run on an arbitrarily chosen path until its completion.

LTL model checking. The $\mathbb{K}$ runner includes a hook to the Maude LTL (Linear Temporal Logic) model checker [9]. Thus, one can model check LTL formulas on programs having a finite state space (or by restricting the verification to a finite subset of the state space). This requires an extension of the syntax and semantics of a language for including labels that are used as atomic propositions in the LTL formulas. Assertions (predicates) on the program's variables can be used as propositions in the formulas as well, using the approach outlined in

[22]. For instance, a guess for a while-loop invariant of the program `log.imp` in Figure 5 can be expressed by an atomic proposition:

$$\mathit{Prop} ::= \mathit{logInv}\ (\ \mathit{Id}\ ,\ \mathit{Id}\ ,\ \mathit{Id}\ )$$

having the following semantics:

$$\begin{array}{l}\langle\langle\_\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \models_{Ltl} \mathit{logInv}(A, X, K) \\ \Rightarrow \\ \mathit{lookup}(X, E) * 2^{\mathit{lookup}(X,K)} \leq_{Int} \mathit{lookup}(X, A) \wedge \\ \mathit{lookup}(X, A) <_{Int} (\mathit{lookup}(X, E) +_{Int} 1) * 2^{\mathit{lookup}(X,K)}\end{array}$$

where $\alpha \triangleq \mathit{logInv}(\mathtt{a}, \mathtt{x}, \mathtt{k})$ is equivalent to the assertion $\mathtt{x}{*}2^{\mathtt{k}} \leq \mathtt{a} < (\mathtt{x}{+}1){*}2^{\mathtt{k}}$. The relation $\pi \models_{Ltl} \alpha$ expresses the fact that the current configuration π satisfies the assertion α. The labels of the statements can be seen as LTL atomic propositions as well:

$$\mathit{Prop} ::= \mathit{Id}$$

with the semantics:

$$\langle\langle L\ :\ S \curvearrowright \_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \models_{Ltl} L \Rightarrow \mathit{true}$$

i.e., $\pi \models_{Ltl} L$ whenever the first statement in the cell `k` (that follows to be executed) is labelled with the label L.

We have enriched the $\mathbb{K}$ definition of IMP with syntax and semantics for LTL support. Consider for instance the program `log.imp` in Figure 5, which computes the integer binary logarithm of an integer read from the input. We prove that whenever the loop visits the label `L`, the invariant $\mathit{logInv}(\mathtt{a}, \mathtt{x}, \mathtt{k})$ holds. The while-loop invariant can be checked for concrete executions given by concrete input values:

```
$ krun log.imp -cPC="true" -cIN="10" -ltlmc "□Ltl (L →Ltl logInv(a, x, k))"
$ true
```

Above the symbolic version of the language is used for concrete inputs; this is possible because the only feasible executions in this case are the concrete ones.

However, when the input is symbolic, then the evaluation of the LTL atomic propositions becomes tricky because the relation $\models_{Ltl}$ is now between symbolic configurations and atomic propositions: $\pi \wedge \phi \models_{Ltl} \alpha$ and its evaluation depends on the path condition ϕ. A correct solution for such cases is to split the executions into two branches: one with the path condition $\phi \wedge \alpha$ and the other one for the case $\phi \wedge \neg\, \alpha$. This is not practically reasonable when we work with an external model-checker because at least of the following reasons: 1) the set of assertions α is infinite; 2) if we restrict the approach only to atomic propositions occurring in the checked formula, we have to recompile the definition each time a new formula is checked; 3) the state space could explode more. We opted for a more experimental solution, that works fine whenever the model-checker returns a positive answer: $\pi \wedge \phi \models_{Ltl} \alpha$ holds whenever $\phi \wedge \neg\, \alpha$ is unsatisfiable, i.e., $\phi \longrightarrow \alpha$ is valid. This covers the cases when all concrete configurations in $[\![\pi \wedge \phi]\!]$ satisfies α. This means that if an LTL formula holds for all symbolic executions, then it holds for all concrete executions covered by the symbolic ones. If the formula does not hold and a counter-example is reported, then the concrete executions covered by the counter-example must be analysed. For instance, checking the program

```
int x; x = read(); x = x + 1;
```

against $\Diamond_{Ltl}\, odd(\mathtt{x})$ and the symbolic input a returns a counterexample because the prover cannot deduce that one of a or $a +_{Int} 1$ satisfies $odd(\mathtt{x})$, which holds whenever the value of the program variable `x` given as argument is odd.

Another issue with the LTL model-checking is given by the fact that the $\mathbb{K}$ tool usually implements only an approximation of the language definition, see [4] for a detailed discussion on this aspect. This approximation is propagated to the symbolic version and therefore the results of an LTL model-checking must be interpreted with respect this approximation.

Here we describe how the LTL model-checker can be called to check formulas over the the space of symbolic executions using the experimental prototype, which can be accessed with the online interface `https://fmse.info.uaic.ro/tools/Symbolic/`. We let `a` be a symbolic value and restrict it in the interval (0..10) to obtain a finite state space. We prove that the above property, denoted by `logInv(a,x,k)` holds whenever the label `L` is visited and `a` is in the given interval, using the following command (again, slightly edited for better readability):

```
$ krun log.imp -cPC="a >Int 0  ∧Bool  a <Int 10" -cIN="a"
               -ltlmc "□Ltl (L →Ltl logInv(a, x, k))"
```

The $\mathbb{K}$ runner executes the command by calling the Maude LTL model-checker for the LTL formula $\Box_{Ltl}\,(\mathtt{L} \rightarrow_{Ltl} \mathtt{logInv}(\mathtt{a},\mathtt{x},\mathtt{k}))$ and the initial configuration having the program `log.imp` in the computation cell `k`, the symbolic value a in the input cell in, and the constraint $a >_{Int} 0 \wedge_{Bool} a <_{Int} 10$ in the path condition. The result returned by the tool is that the above LTL formula holds.

7.2.2. SIMPLE*, symbolic arrays, and bounded model checking*

We illustrate symbolic arrays in the SIMPLE language and show how the $\mathbb{K}$ runner can directly be used for performing bounded model checking. In the program in Figure 6, the `init` method assigns the value `x` to the array `a` at an index `j`, then fills the array with ascending even numbers until it encounters x in the array; it prints *error* if the index `i` went beyond `j` in that process. The array and the indexes `i`, `j` are parameters to the function, passed to it by the `main` function which reads them from the input. In [1] it has been shown, using model-checking and abstractions on arrays, that this program never prints *error*.

We obtain the same result by running the program with symbolic inputs and using the $\mathbb{K}$ runner as a bounded model checker:

```
void init(int[] a, int x, int j){            void main() {
     int i = 0, n = sizeOf(a);                    int n = read();
     a[j] = x;                                    int j = read();
     while (a[i] != x && i < n) {                 int x = read();
       a[i] = 2 * i;                              int a[n], i = 0;
       i = i + 1;                                 while (i < n) {
     }                                              a[i] = read();
     if (i > j) {                                   i = i + 1;
       print("error");                            }
     }                                            init(a, x, j);
}                                            }
```

Figure 6: SIMPLE program: `init-arrays`

```
$ krun init-arrays.simple -cPC="n >Int 0" -search -cIN="n j x a1 a2 a3"
                          -pattern="<T> <out> error </out> B:Bag </T>"
Search results:
No search results
```

The initial path condition is $n >_{Int} 0$. The symbolic inputs for `n,j,x` are entered as $n\ j\ x$, and the array elements $a1\ a2\ a3$ are also symbolic. The `-pattern` option specifies a pattern to be searched in the final configuration: the text *error* should be in the configuration's output buffer. The above command thus performs a bounded model-checking with symbolic inputs (the bound is implicitly set by the number of array elements given as inputs - 3). It does not return any solution, meaning that that the program will never print *error*.
The result was obtained using symbolic execution without any additional tools or techniques. We note that array sizes are symbolic as well, a feature that, to our best knowledge, is not present in other symbolic execution frameworks.

7.2.3. KOOL*: testing virtual method calls on lists*

Our last example (Figure 7) is a program in the KOOL object-oriented language. It implements lists and ordered lists of integers using arrays. We use symbolic execution to check the well-known virtual method call mechanism of object-oriented languages: the same method call, applied to two objects of different classes, may have different outcomes.

The `List` class implements (plain) lists. It has methods for creating, copying, and testing the equality of lists, as well as for inserting and deleting elements in a list. Figure 7 shows only a part of them. The class `OrderedList` inherits from `List`. It redefines the `insert` method in order to ensure that the sequences of elements in lists are sorted in increasing order. The `Main` class creates a list `l1`, initializes `l1` and an integer variable `x` with input values, copies `l1` to a list `l2` and then inserts and deletes `x` in `l1`. Finally it compares `l1` to `l2` element by element, and prints *error* if it finds them different. We use symbolic execution to show that the above sequence of method calls results in different outcomes, depending on whether `l1` is a `List` or an `OrderedList`. We first try the case where `l1` is a `List`, by issuing the following command to the $\mathbb{K}$ runner:

```
$ krun  lists.kool   -search -cIN="e1 e2 x"
                     -pattern="<T> <out> error </out> B:Bag </T>"
 Solution 1, State 50:
<path-condition>
```

```
class List {
  int a[10];
  int size, capacity;
  ...

  void insert (int x) {
    if (size < capacity) {
      a[size] = x;  ++size;
    }
  }

  void delete(int x) {
    int i = 0;
    while(i < size-1 && a[i] != x) {
      i = i + 1;
    }
    if (a[i] == x) {
      while (i < size - 1) {
        a[i] = a[i+1];
        i = i + 1;
      }
      size = size - 1;
    }
  }
  ...
}
```

```
class OrderedList extends List {
  ...
  void insert(int x){
    if (size < capacity) {
      int i = 0, k;
      while(i < size && a[i] <= x) {
        i = i + 1;
      }
      ++size; k = size - 1;
      while(k > i) {
        a[k] = a[k-1]; k = k - 1;
      }
      a[i] = x;
    }
  }
}
class Main {
  void Main() {
    List l1 = new List();
    ... // read elements of l1 and x
    List l2 = l1.copy();
    l1.insert(x); l1.delete(x);
    if (l2.eqTo(l1) == false) {
      print("error\n");
    }
  }
}
```

Figure 7: `lists.kool`: implementation of lists in KOOL

$e_1 \;=\; x \;\wedge_{Bool}\; \neg_{Bool}\; (e_1 \;=\; e_2)$
`</path-condition>`
`...`

The command initializes `l1` with two symbolic values (e_1, e_2) and sets `x` to the symbolic value x. It searches for configurations that contain *error* in the output. The tool finds one solution, with $e_1 = x$ and $e_1 \neq e_2$ in the path condition. Since `insert` of `List` appends x at the end of the list and deletes the first instance of x from it, `l1` consists of (e_2, x) when the two lists are compared, in contrast to `l2`, which consists of (e_1, e_2). The path condition implies that the lists are different.

The same command on the same program but where `l1` is an `OrderedList` finds no solution. This is because `insert` in `OrderedList` inserts an element in a unique place (up to the positions of the elements equal to it) in an ordered list, and `delete` removes either the inserted element or one with the same value. Hence, inserting and then deleting an element leaves an ordered list unchanged.

Thus, virtual method call mechanism worked correctly in the tested scenarios. An advantage of using our symbolic execution tool is that the condition on the inputs that differentiated the two scenarios was discovered by the tool. This feature can be exploited in other applications such as test-case generation.

The examples illustrated in this section are meant to exhibit some features of our implementation, but also to show that our approach is language independent. The $\mathbb{K}$ definitions of the three languages that we use (i.e., IMP, SIMPLE, and KOOL) have different sizes and they capture some essential features that can be found in programming languages today. Besides these examples, our tool can be used for real-life language definitions too (see, for instance, PHP [15]).

Another advantage of our approach is that it is formal. The coverage and precision properties ensure that results of different analyses performed on symbolic executions also hold for concrete executions. On the downside, some programs cause efficiency issues because of the large number of states to explore. In order to reduce the state space, we have implemented a mechanism which allows users to choose which semantical rules generate new states (see Section 7.3).

7.3. The implementation of the tool

Our tool was developed as an extension of the $\mathbb{K}$ 3.5 compiler. A part of the connection to the Z3 SMT solver was done in $\mathbb{K}$ itself, and the rest of the code is written in Java. The $\mathbb{K}$ compiler (`kompile`) is organized as a list of transformations applied to the abstract syntax tree of a $\mathbb{K}$ definition. Our compiler inserts additional transformations (formally described in Section 5.3 and Section 6.2) when the $\mathbb{K}$ compiler is called with the `-backend symbolic` option.

The compiler adds syntax declarations for each sort, which allows users to use symbolic values written as, e.g., `#symSort(x)` in their programs. The tool also generates predicates used to distinguish between concrete and symbolic values.

For handling the path condition, a new configuration cell, `<path-condition>` is automatically added to the configuration. The transformations of rules discussed in Subsection 5.3 are also implemented as transformers applied to rules. There is a transformer for linearizing rules, which collects all the variables that appear more than once in the left hand side of a rule, generates new variables for each one, and adds an equality in the side condition. There is also a transformer that replaces data subterms with variables, following the same algorithm as the previous one, and a transformer that adds rule's conditions in the symbolic configuration's path conditions. In practice, building the path condition blindly may lead to exploration of program paths which are not feasible. Therefore, at each step, the tool has to check whether the next symbolic state is satisfiable and it does that by connecting to the Z3 SMT solver. For this reason, the transformer that collects the path condition also adds, as a side condition to $\mathbb{K}$ rules, a call to the SMT solver of the form `checkSat(`ϕ`)` $\neq$ `"unsat"`, where the `checkSat` function calls the SMT solver over the current path condition ϕ. When the path condition is found unsatisfiable the current path is not explored any longer. Note that we use `checkSat(`ϕ`)` $\neq$ `"unsat"` instead of `checkSat(`ϕ`)` $=$ `"sat"` because we do not want to miss possibly feasible executions. This can happen if the SMT solver cannot decide whether ϕ is satisfiable: the condition `checkSat(`ϕ`)` $=$ `"sat"` becomes false, the rule does not apply, and the exploration stops even though the path is not known to be unfeasible for sure. Another problem that arises here is that, in $\mathbb{K}$, the condition of rules may also contain internally generated predicates needed only for matching. Those predicates should not be part of the path condition, therefore they had to be filtered out from rule's conditions before the latter are added to path conditions.

Not all the rules from a $\mathbb{K}$ definition must be transformed. This is the case, e.g., of the rules computing functions or predicates. We have created a transformer that detects such rules and marks them with a tag. The tag can also be used by the user, in order to prevent the transformation of other rules if needed. Finally, in order to allow passing symbolic inputs to programs we generated a variable `$IN`, initialized at runtime by `krun` with the value of the option `-cIN`.

Regarding performances, our generic and formal tool is, quite understandably, not in the same league as existing pragmatic tools, which are dedicated to specific languages (e.g. Java PathFinder for Java, PEX for C#, KLEE for LLVM) and are focused on specific applications of symbolic execution. Our purpose is to automatically generate, from a formal definition of any language, a symbolic semantics capable of symbolically executing programs in that language, and to provide users with means for building their applications on top of our tool. For instance, our symbolic execution was used in combination with the $\mathbb{K}$ model-checker for verifying some LTL properties over PHP programs [15]. Formal verification of programs based on deductive methods is also currently being built on top of our tool [21].

8. Conclusion and Future Work

In this paper we present a language-independent approach to symbolic execution. The approach is based on language transformations. Starting from the formal definition of a language $\mathcal{L}$, whose operational semantics is defined by rewriting (specifically, by Reachability Logic rules [30]), a so-called symbolic language $\mathcal{L}^{\mathfrak{s}}$ is constructed, by changing the model underlying $\mathcal{L}$ into a *symbolic model*, and by adapting the semantical rules to compute over the symbolic model. The symbolic model consists of equivalence classes of formulas of Matching Logic [30] denoting possibly infinite sets of concrete program states.

The symbolic execution of programs in $\mathcal{L}$ is defined to be the (usual, i.e., concrete) execution of the corresponding programs in $\mathcal{L}^{\mathfrak{s}}$. We prove that the notion of symbolic execution thus defined has the properties of *coverage*, meaning that for each concrete execution there is corresponding feasible symbolic one on the same path of instructions, and *precision*, meaning that each feasible symbolic execution has a corresponding concrete execution on the same path.

These theoretical results have practical consequences. They ensure that results of, e.g., reachability analyses or model checking performed on symbolic executions (a natural thing to do, since one symbolic execution encodes a possibly infinite sets of concrete executions) also hold on concrete program executions (which are the executions one is ultimately interested in analysing). We have implemented our approach in the $\mathbb{K}$ framework and have applied to the model checking and reachability analysis of programs from several languages.

The key difference between our work and existing ones is that our approach is both formal and generic. There are many language-specific symbolic execution approaches and tools, a relevant sample of which is presented in the related works section. By being language-specific, tools can obtain high performances on programs of the language in question; but when a new version of the language arrives, the tool may become obsolete. Tools not formally grounded on formal semantics can also obtain high performances; but to trust their analyses one has to trust that the tool implicitly implements a correct language semantics.

Our approach focuses on genericity and formality, in order to avoid the above-mentioned drawbacks of language-specific and/or semantics-agnostic tools.

Future Work. We are planning to use symbolic execution as the basic verification mechanism for program logics also developed in the $\mathbb{K}$ framework (such as reachability logic [30]. We have already made some progress in this direction, by proposing a procedure which uses symbolic execution for verifying Reachability Logic specifications [21]. More generally, our symbolic execution approach can be used for program testing, debugging, and verification, following the ideas presented in related work, but with the added value of being language independent and being grounded in formal operational semantics.

Acknowledgements. The results presented in this paper would not have been possible without the valuable support from the $\mathbb{K}$ tool development team (`http://k-framework.org`). The work presented here was supported in part by Romanian Contracts 161/15.06.2010, SMIS-CSNR 602-12516 (DAK) and POSDRU/159/1.5/S/137750, and by Région Nord Pas-de-Calais, via a CPER Contract with Inria Lille Nord-Europe.

References

[1] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. In *Proceedings of the Workshop on Software Model Checking*, volume 144 - 3, pages 79 – 94, 2006.

[2] A. Arusoaie, D. Lucanu, and V. Rusu. A generic framework for symbolic execution. In *6th International Conference on Software Language Engineering*, volume 8225 of *LNCS*, pages 281–301. Springer Verlag, 2013. Also available as a technical report `http://hal.inria.fr/hal-00853588`.

[3] A. Arusoaie, D. Lucanu, and V. Rusu. Towards a semantics for {OCL}. *Electronic Notes in Theoretical Computer Science*, 304(0):81 – 96, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[4] A. Arusoaie, D. Lucanu, V. Rusu, T. F. Serbanuta, A. Stefanescu, and G. Rosu. Language definitions as rewrite theories. In *Proceedings of the 10th International Workshop on Rewriting Logic and its Applications (WRLA'14)*, volume 8663 of *LNCS*, pages 97–112. Springer, 2014.

[5] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[7] D. Bogdănaş and G. Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, 2015.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

[9] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

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

[11] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

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

[13] C. Ellison and G. Rosu. An executable formal semantics of c with applications. In *ACM SIGPLAN Notices*, volume 47, pages 533–544. ACM, 2012.

[14] S. Escobar, J. Meseguer, and R. Sasse. Variant narrowing and equational unification. *Electr. Notes Theor. Comput. Sci.*, 238(3):103–119, 2009.

[15] D. Filaretti and S. Maffeis. An executable formal semantics of PHP. In *Proceedings of European Conference on Object-Oriented Programming*, 2014.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[17] M. Hills and G. Rosu. Kool: An application of rewriting logic to language prototyping and analysis. In *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 246–256. Springer, 2007.

[18] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

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

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

[21] D. Lucanu, V. Rusu, A. Arusoaie, and D. Nowak. Verifying reachability-logic properties on rewriting-logic specifications. *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, to appear. Also available as a technical report `http://www.infoiasi.ro/~tr/tr.pl.cgi`.

[22] D. Lucanu, T. F. Şerbănuţă, and G. Roşu. The K Framework distilled. In *9th International Workshop on Rewriting Logic and its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012. Invited talk.

[23] J. Meseguer and P. 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.

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

[25] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *STTT*, 11(4):339–353, 2009.

[26] C. Pecheur, J. Andrews, and E. D. Nitto, editors. *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010.* ACM, 2010.

[27] C. Rocha, J. Meseguer, and C. A. Muñoz. Rewriting modulo SMT and open system analysis. In *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2014.

[28] G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.

[29] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[30] G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 555–574. ACM, 2012.

[31] G. Rosu and A. Stefanescu. From hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.

[32] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2012.

[33] P. H. Schmitt and B. Weiß. Inferring invariants by symbolic execution. In *Proceedings of 4th International Verification Workshop (VERIFY'07)*, 2007.

[34] K. Sen, D. Marinov, and G. 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.

[35] T. F. Serbanuta, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Rosu. The K primer (version 2.5). In M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, to appear.

[36] T.-F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

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

[38] M. Staats and C. S. Păsăreanu. Parallel symbolic execution for structural test generation. In P. Tonella and A. Orso, editors, *ISSTA*, pages 183–194. ACM, 2010.

[39] K. 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.