A Generic Framework for Symbolic Execution

Andrei Arusoaie[1], Dorel Lucanu[1], and Vlad Rusu[2]

[1] Faculty of Computer Science, Alexandru Ioan Cuza University, Iaşi, Romania
andrei.arusoaie@gmail.com, dlucanu@info.uaic.ro
[2] Inria Lille Nord Europe, France
vlad.rusu@inria.fr

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 automatically generated, which has the same syntax as the original one but whose semantics extends data domains with symbolic values and adapts semantical rules to deal with these values. Then, the symbolic execution of concrete programs is the execution of programs with the new symbolic semantics, on symbolic input data. We prove that the symbolic execution thus defined has the properties naturally expected from it. A prototype implementation of our approach was developed in the $\mathbb{K}$ Framework. We demonstrate the genericity of our tool by instantiating it on several languages, and show how it can be used for the symbolic execution and model checking of several programs.

1 Introduction

Symbolic execution is a well-known program analysis technique introduced in 1976 by James C. King [12]. Since then, it has proved its usefulness for testing, verifying, and debugging programs. Symbolic execution consists in executing programs with symbolic inputs, instead of concrete ones, and it involves the processing of expressions involving symbolic values [19]. The main advantage of symbolic execution is that it allows reasoning about multiple concrete executions of a program, and its main disadvantage is the state-space explosion determined by decision statements and loops. Recently, the technique has found renewed interest in the formal-methods community due to new algorithmic developments and progress in decision procedures. Current applications of symbolic execution are diverse and include automated test input generation [13], [27], invariant detection [18], model checking [11], and proving program correctness [26,7]. We believe there is a need for a formal and generic approach to symbolic execution, on top of which language-independent program analysis tools can be developed.

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

M. Erwig, R.F. Paige, and E. Van Wyk (Eds.): SLE 2013, LNCS 8225, pp. 281–301, 2013.

the control flow of a program is determined by symbolic values (e.g., the next instruction to be executed is a conditional statement, 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]. To our best knowledge, our framework is the only one supporting automatic derivation of the symbolic semantics of languages from their concrete semantics. On the theoretical side, we introduce a transformation between languages such that the symbolic execution in the source language is defined as the concrete execution in the transformed language. We prove that the symbolic execution thus defined has the following properties, which ensure that it is related to concrete program 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.

On the practical side, we present a prototype implementation of our approach in $\mathbb{K}$ [20], a framework dedicated to defining formal operational semantics of languages. Developing our tool within the $\mathbb{K}$ framework enables us to benefit from the many existing language definitions written in $\mathbb{K}$. We briefly describe our implementation as a language-engineering tool, and demonstrate its genericity by instantiating it on several nontrivial languages defined in $\mathbb{K}$. We emphasize 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 program execution as well as Linear Temporal Logic model checking and bounded model checking using our tool.

We note that the proposed approach deals with symbolic data, not with symbolic code. Hence, it is restricted to languages in which data and code are distinct entities that cannot be mixed. This excludes, for example, higher-order functional languages in which code can be passed as data between functions.

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 [28] 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

[1] Most existing operational semantics styles (small-step, big-step, reduction with evaluation contexts, ...) have been shown to be representable in this way in [25].

multiple decision procedures. We have instantiated our generic approach to a formal definition of Java defined in the $\mathbb{K}$ framework, and have performed symbolic execution on several programs. This shows that our tool can tackle real languages.

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 [9], CUTE [23], EXE [4], PEX [5]. 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 [12]. It can also be used for proving program correctness. There are several tools (e.g. Smallfoot [3]) which use symbolic execution together with separation logic to prove Hoare triples. There are also approaches that attempt to automatically detect invariants in programs([18], [22]). 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 [17]. Such analyses rely on powerful unification-modulo-theories algorithms [8], 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.

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, in order to generate tests for programs, the only thing that has to be added to our framework is to request models of path conditions using, e.g., SMT solvers. Formal verification of programs based on deductive methods and predicate abstractions are also currently being built on top of our tool.

Structure of the Paper. Section 2 introduces our running example (the simple 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 shows how the definition of a language $\mathcal{L}$ can be automatically transformed into the definition

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

Fig. 1. $\mathbb{K}$ Syntax of IMP

of a language $\mathcal{L}^{\mathfrak{s}}$ by extending the data of $\mathcal{L}$ with symbolic values, and by providing the semantical rules of $\mathcal{L}$ with means to process those values. Section 5 deals with the symbolic semantics and with its relation to the concrete semantics, establishing the coverage and precision results stated in this introduction. Section 6 describes an implementation of our approach in the $\mathbb{K}$ framework and show how it is automatically instantiated to nontrivial languages defined in $\mathbb{K}$. An Appendix (for the reviewers only, not to be included in the final version) contains more detailed descriptions of the examples and of the tool.

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

Our running example is IMP, a simple imperative language intensively used in research papers. The syntax of IMP is described in Figure 1 and is mostly self-explainatory 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.

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

Fig. 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 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

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

Fig. 3. $\mathbb{K}$ Semantics of IMP

`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 set 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:

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

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

3 The Ingredients of a Language Definition

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

1. A many-sorted algebraic signature Σ, which includes at least a sort *Cfg* for *configurations* and a sort *Bool* for *constraint formulas*. For the sake of presentation, we assume in this paper that the constraint formulas are Boolean terms built with a subsignature $\Sigma^{\mathsf{Bool}} \subseteq \Sigma$ including the boolean constants and operations. Σ may also include other subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all *data* sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{\mathsf{Data}}$. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise infinite set of variables *Var*, let $T_\Sigma(Var)$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(Var)$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t.
2. A Σ^{Data}-model $\mathcal{D}$, which interprets the data sorts and operations. We assume that the model $\mathcal{D}$ is *reachable*, i.e., for all $d \in \mathcal{D}$ there exists a term $t \in T_{\Sigma^{\mathsf{Data}}}$ such that $d = \mathcal{D}_t$. Let $\mathcal{T} \triangleq \mathcal{T}(\mathcal{D})$ denote the free Σ-model generated by $\mathcal{D}$, i.e., $\mathcal{T}$ interprets the non-data sorts as ground terms over the signature
$$(\Sigma \setminus \Sigma^{\mathsf{Data}}) \cup \bigcup_{d \in Data} \mathcal{D}_d \tag{1}$$
where $\mathcal{D}_d$ denotes the carrier set of the sort d in the algebra $\mathcal{D}$, and the elements of $\mathcal{D}_d$ are added to the signature $\Sigma \setminus \Sigma^{\mathsf{Data}}$ as constants of sort d. The satisfaction relation $\rho \models b$ between valuations ρ and constraint formulas $b \in T_{\Sigma,Bool}(Var)$ is defined by $\rho \models b$ iff $\rho(b) = \mathcal{D}_{true}$. For simplicity, we often write in the sequel $true, false, 0, 1 \ldots$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}, \mathcal{D}_0, \mathcal{D}_1, \ldots$
3. A set $\mathcal{S}$ of rewrite rules. Each rule is a pair of the form $l \wedge b \Rightarrow r$, where $l, r \in T_{\Sigma,Cfg}(Var)$ are the rule's *left-hand-side* and the *right-hand-side*, respectively, and $b \in T_{\Sigma,Bool}(Var)$ is the *condition*. The formal definitions for rules and for the transition system defined by them are given below.

We explain these concepts on IMP. Nonterminals in the syntax (*Id*, *Int*, *Bool*, ...) are sorts in Σ. Each production from the syntax defines an operation in Σ; e.g, the production $AExp ::= AExp \texttt{ + } AExp$ defines the operation $\_+\_ : AExp \times AExp \to AExp$. These operations define the constructors of the result sort. For the sort *Cfg*, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : Code \times Map_{Id,Int} \to Cfg$. The expression $\langle\langle X := I \curvearrowright C\rangle_{\mathsf{k}}\langle X \mapsto 0\ Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a term of $T_{Cfg}(Var)$, where X is a variable of sort *Id*, I is a variable of sort *Int*, C is a variable of sort *Code* (the rest of the computation), and *Env* is a variable of sort $Map_{Id,Int}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets *Int* as the set of integers, the operations like $+_{Int}$ (cf. Figure 3) as the corresponding usual operation on integers, *Bool* as the set of Boolean values $\{false, true\}$, the operation like $\wedge$ as

the usual Boolean operations, the sort $Map_{Id,Int}$ as the set of maps $X \mapsto I$, where X ranges over identifiers Id and I over the integers. The value of an identifier X is an environment M is $lookup(X, M)$, and the environment M, updated by binding an identifier X to a value I, is $update(X, M, I)$. Here, $lookup()$ and $update()$ are operations in a signature $\Sigma^{\mathsf{Map}} \subseteq \Sigma^{\mathsf{Data}}$ of maps. The other sorts, *AExp*, *BExp*, *Stmt*, and *Code*, are interpreted in the algebra $\mathcal{T}$ as ground terms over a modification of the form (1) of the signature Σ, in which data subterms are replaced by their interpretations in $\mathcal{D}$. For instance, the term `if` $1 >_{Int}$ 0 `then skip else skip` is interpreted as `if` $\mathcal{D}_{true}$ `then skip else skip`.
We now formally introduce the notions required for defining semantical rules.

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

A basic pattern π defines a set of (concrete) configurations, and the condition b gives additional constraints these configurations must satisfy.
Remark 1. The above definition is a particular case of a definition in [21]. There, a pattern is a first-order logic formula with configuration terms as sub-formulas. In this paper we keep the conjunction notation from first-order logic but separate basic patterns from constraints. Note that first-order formulas can be encoded as terms of sort *Bool*, where the quantifiers become constructors. The satisfaction relation $\models$ is then defined, for such terms, like the usual FOL satisfaction.

We identify basic patterns π with patterns $\pi \wedge true$. Sample patterns are $\langle\langle I_1 + I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \,/\, I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$.

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

4 Symbolic Semantics by Language Transformation

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

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

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

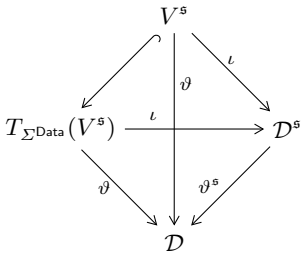

Fig. 4. Diagram Characterising Data Symbolic Domain $\mathcal{D}^{\mathfrak{s}}$

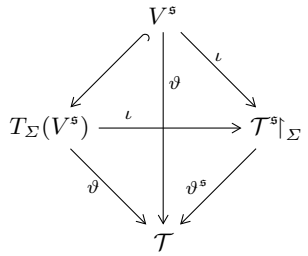

Fig. 5. Lifting Diagram in Fig. 4 to from Data Domain $\mathcal{D}$ to $\mathcal{T}^{\mathfrak{s}}\restriction_{\Sigma}$

4.1 Extending the Signature Σ to a Symbolic Signature $\Sigma^{\mathfrak{s}}$

The signature $\Sigma^{\mathfrak{s}}$ extends Σ with a sort $Cfg^{\mathfrak{s}}$ and a constructor $\langle \_, \_ \rangle : Cfg \times Bool \to Cfg^{\mathfrak{s}}$, which builds symbolic configurations as pairs of configurations over symbolic data and Booleans term denoting path conditions.

Example 1. For the IMP example we enrich the configuration with a new cell:

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

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

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

We first deal with the *symbolic domain* $\mathcal{D}^{\mathfrak{s}}$, a Σ^{Data}-algebra with the following properties:

1. The Σ^{Data}-algebra $\mathcal{D}$ is a sub-algebra of $\mathcal{D}^{\mathfrak{s}}$.
2. We assume an infinite, sort-wise set of *symbolic values* $V^{\mathfrak{s}}$ of the data sorts, disjoint from *Var* and from symbols in Σ, and assume that there is an injection $\iota : V^{\mathfrak{s}} \to \mathcal{D}^{\mathfrak{s}}$ such that for any valuation $\vartheta : V^{\mathfrak{s}} \to \mathcal{D}$ there exists a unique algebra morphism $\vartheta^{\mathfrak{s}} : \mathcal{D}^{\mathfrak{s}} \to \mathcal{D}$ such that the diagram in Figure 4 commutes. The diagram essentially says that the interpretation of terms like $a^{\mathfrak{s}} +_{Int} b^{\mathfrak{s}}$ via ϑ is the same as that given by the composition of ι with $\vartheta^{\mathfrak{s}}$.
3. The satisfaction relation $\models$ is extended to constraint formulas $\phi^{\mathfrak{s}} \in \mathcal{D}^{\mathfrak{s}}_{Bool}$ and valuations $\vartheta : V^{\mathfrak{s}} \to \mathcal{D}$ such that $\vartheta \models \phi^{\mathfrak{s}}$ iff $\vartheta^{\mathfrak{s}}(\phi^{\mathfrak{s}}) = \mathcal{D}_{true}$.

For instance, $\mathcal{D}^{\mathfrak{s}}$ can be the algebra of ground terms over the signature $\Sigma^{\mathsf{Data}}(V^{\mathfrak{s}} \cup \mathcal{D})$, or the quotient of this algebra modulo the congruence defined by some set of equations (which can be used in practice as simplification rules).

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

By the definition of $\mathcal{T} = \mathcal{T}(\mathcal{D})$, there is a unique Σ-morphism $\mathcal{T} \to \mathcal{T}(\mathcal{D}^{\mathfrak{s}})$. We note that the extended definition $(\Sigma, \mathcal{S}, \mathcal{T}(\mathcal{D}^{\mathfrak{s}}))$ is not suitable for symbolic executions because the symbolic values in $V^{\mathfrak{s}}$ are constrained by the computations and decisions taken up to that point. This is why we extended the signature to $\Sigma^{\mathfrak{s}}$, in which the path condition becomes a component of the configuration.

Next, we naturally define the model $\mathcal{T}^{\mathfrak{s}}$ as being the free $\Sigma^{\mathfrak{s}}$-model generated by $\mathcal{D}^{\mathfrak{s}}$. Since there is an inclusion signature morphism $\Sigma \hookrightarrow \Sigma^{\mathfrak{s}}$, $\mathcal{T}^{\mathfrak{s}}$ can also be

seen as a Σ-model $\mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$, where only the interpretations of the symbols from Σ are considered. This allows us to lift up the diagram in Figure 4 at the level of the model $\mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$ and in particular to define $\vartheta^{\mathfrak{s}} : \mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma} \to \mathcal{T}$ as the unique function from $\mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$ to $\mathcal{T}$ that makes the diagram in Figure 5 commute. Furthermore, Σ and $\Sigma^{\mathfrak{s}}$ have the same data sub-signature and $\mathcal{D}$ is a sub-algebra of $\mathcal{D}^{\mathfrak{s}}$, hence there is a unique Σ-morphism $\mathcal{T} \to \mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$. All these properties of the model $\mathcal{T}^{\mathfrak{s}}$ show that it is a suitable model for both concrete and symbolic executions.

However, the semantical rules $\mathcal{S}$ still have to be transformed into rules on symbolic configurations including path conditions. Moreover, we must ensure that the transition system defined by the new rules has the properties of coverage and precision with respect to the transition system defined by $(\Sigma, \mathcal{S}, \mathcal{T})$. This requires some transformations of the rules $\mathcal{S}$, to be presented later in the paper.

The following lemma is crucial for obtaining symbolic executions via matching.

Lemma 1 (Semantic Unification is Reduced to Matching). *Let us consider $l \in T_\Sigma(Var)$, $\rho : Var \to \mathcal{T}$, $\pi^{\mathfrak{s}} \in \mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$, $\vartheta : V^{\mathfrak{s}} \to \mathcal{T}$ such that l is linear, any data sub term of l is a variable, and $\rho(l) = \vartheta^{\mathfrak{s}}(\pi^{\mathfrak{s}})$ (i.e., l and $\pi^{\mathfrak{s}}$ are semantically unifiable in $\mathcal{T}$). Then there is a (symbolic) valuation $\sigma : Var \to \mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$ such that $\sigma(l) = \pi^{\mathfrak{s}}$ and $\vartheta^{\mathfrak{s}}(\sigma(x)) = \rho(x)$ for each $x \in Var$.*

Proof. We first prove the slightly weaker property ($\diamondsuit$): there exists a valuation $\sigma : var(l) \to \mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$ such that $\sigma(l) = \pi^{\mathfrak{s}}$ and $\vartheta^{\mathfrak{s}}(\sigma(x)) = \rho(x)$ for each $x \in var(l)$.

To prove ($\diamondsuit$) we proceed by structural induction on l. If l is a variable x, then we take $\sigma(x) = \pi^{\mathfrak{s}}$ and the conclusion of the lemma is obviously satisfied. We assume now that $l = f(l_1, \ldots, l_n)$, $n \geq 0$. The result sort of f is a non-data sort by the hypotheses, hence $\mathcal{T}_f(a_1, \ldots, a_n) = f(a_1, \ldots, a_n)$ and $\mathcal{T}^{\mathfrak{s}}_f(b_1, \ldots, b_n) = f(b_1, \ldots, b_n)$ by the definition of $\mathcal{T}$ and $\mathcal{T}^{\mathfrak{s}}$, respectively. Consequently, $\rho(l) = f(\rho(l_1), \ldots, \rho(l_n))$, $\pi^{\mathfrak{s}} = f(\pi^{\mathfrak{s}}_1, \ldots, \pi^{\mathfrak{s}}_n))$, $\vartheta^{\mathfrak{s}}(\pi^{\mathfrak{s}}) = f(\vartheta^{\mathfrak{s}}(\pi^{\mathfrak{s}}_1), \ldots, \vartheta^{\mathfrak{s}}(\pi^{\mathfrak{s}}_n))$, and $\rho(l_i) = \rho^{\mathfrak{s}}(\pi^{\mathfrak{s}}_i)$, $i = 1, \ldots, n$, for certain $\pi^{\mathfrak{s}}_1, \ldots, \pi^{\mathfrak{s}}_n \in T^{\mathfrak{s}}{\restriction_\Sigma}$. Recall that for each sort s in Σ, $(T^{\mathfrak{s}}{\restriction_\Sigma})_s = \mathcal{T}^{\mathfrak{s}}_s$. Each term l_i preserves the properties of l, hence there is σ_i satisfying the conclusion of lemma for l_i and $\pi^{\mathfrak{s}}_i$, i.e. $\sigma_i(l_i) = \pi^{\mathfrak{s}}_i$ and $\rho(x) = \vartheta^{\mathfrak{s}}(\sigma_i(x))$ for each $x \in var(l_i)$. Since l is linear, $var(l) = var(l_1) \uplus \ldots \uplus var(l_n)$. It follows we may define $\sigma : var(l) \to \mathcal{T}^{\mathfrak{s}}{\restriction_\Sigma}$ such that $\sigma(x) = \sigma_i(x)$ iff $x \in var(l_i)$. We have $\sigma(l) = f(\sigma(l_1), \ldots, \sigma(l_n)) = f(\sigma_1(l_1), \ldots, \sigma_n(l_n)) = f(\pi^{\mathfrak{s}}_1, \ldots, \pi^{\mathfrak{s}}_n)) = \pi^{\mathfrak{s}}$. The property $\rho(x) = \vartheta^{\mathfrak{s}}(\sigma(x))$ for each $x \in var(l)$ is inherited from σ_i.

The prove the lemma, we need to extend the valuation σ to Var such that $\vartheta^{\mathfrak{s}}(\sigma(x)) = \rho(x)$ for all $x \in Var$, using the reachability of the data domain $\mathcal{D}$:

- first, we prove that the function $\vartheta^{\mathfrak{s}} : \mathcal{T}{\restriction_\Sigma} \to \mathcal{T}$ is surjective. For this, consider any $\tau \in \mathcal{T}$, thus, $\tau \triangleq C[\tau_1, \ldots \tau_n]$ with $\tau_1, \ldots, \tau_n \in \mathcal{D}$ and C a Σ-context, since $\mathcal{T}$ is the free Σ-model generated by $\mathcal{D}$. Since $\mathcal{D}$ is reachable, $\tau_i = \mathcal{D}_{t_i}$ for some $t_i \in T_{\Sigma^{\mathsf{Data}}}$, $i = 1, \ldots, n$. Then, we have $\vartheta^{\mathfrak{s}}(\iota(C[t_1, \ldots t_n])) = \vartheta(C[t_1, \ldots t_n])$ per the diagram in Figure 5, and since $C[t_1, \ldots t_n] \in T_\Sigma$ we have $\vartheta(C[t_1, \ldots t_n]) = \mathcal{T}_{C[t_1, \ldots t_n]} = \mathcal{T}_t = \tau$ (as $\vartheta : T_\Sigma(V^{\mathfrak{s}}) \to \mathcal{T}$ maps ground terms in $T_\Sigma(\emptyset)$ to their interpretation in $\mathcal{T}$). Thus, for an arbitrary $\tau \in \mathcal{T}$ we found $\mu \triangleq \iota(C[t_1, \ldots t_n])$ satisfying $\vartheta(\mu) = \tau$, i.e., $\vartheta^{\mathfrak{s}} : \mathcal{T}{\restriction_\Sigma} \to \mathcal{T}$ is surjective.
- thus, for each $x \in Var \setminus var(l)$, we choose $\sigma(x)$ s.t. $\vartheta^{\mathfrak{s}}(\sigma(x)) = \rho(x)$. □

Definition 3 (Satisfaction Relation for Configurations). *A concrete configuration* $\gamma \in \mathcal{T}_{Cfg}$ satisfies *a symbolic configuration* $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$, *written* $\gamma \models \langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$, *if there exists* $\vartheta : V^{\mathfrak{s}} \to \mathcal{D}$ *such that* $\gamma = \vartheta^{\mathfrak{s}}(\pi^{\mathfrak{s}})$ *and* $\vartheta^{\mathfrak{s}}(\phi^{\mathfrak{s}}) = true$.

Example 2. Assume b^s is a symbolic value of sort *Bool*. The configuration

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

satisfies the symbolic configuration

$$\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle \triangleq \langle\langle \texttt{if } b^s \texttt{ then skip else skip} \rangle_{\mathsf{k}} \langle . \rangle_{\mathsf{env}} \langle b^s \rangle_{\mathsf{cnd}} \rangle_{\mathsf{cfg}}$$

thanks to any valuation ϑ that maps b^s to *true*.

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

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

1. Linearising Rules A rule is (left) linear if any variable occurs at most once in its left-hand side. A nonlinear rule can always be turned into an equivalent linear one, by renaming the variables occurring several times and adding equalities between the renamed variables and the original ones to the rule's condition. For example, the last rule from the original IMP semantics (Fig. 3) could have been written as a nonlinear rule:

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

To linearise it we just add a new variable, say X', and a condition, $X' = X$:

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

2. Replacing Data Subterms by Variables Let $Dpos(l)$ be the set of positions ω[2] of the term l such that l_ω is a maximal subterm of a data sort. The next step of our rule transformation consists in replacing all the maximal data subterms of l by fresh variables. The purpose of this step is to make rules match any configuration, including the symbolic ones.

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

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

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

Example 3. Consider the following rule for *if* from the IMP semantics:

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

We replace the constant *true* with a Boolean variable B, and add the condition $B = true$:

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

3. Adding Formulas to Configurations and Rules The last transformation step consists in transforming each rule $l \wedge b \Rightarrow r$ in $\mathcal{S}$ obtained after the previous steps, into the following one:

$$\langle l, \psi \rangle \Rightarrow \langle r, \psi \wedge b \rangle \tag{2}$$

[2] For the notion of position in a term and other rewriting-related notions, see, e.g., [2].

where $\psi \in \mathit{Var}$ is a fresh variable of sort *Bool* (i.e. it does not occur in the rules $\mathcal{S}$) and $\langle \_, \_ \rangle$ is the pairing operation in $\Sigma^{\mathfrak{s}}$. This means that when a symbolic transition is performed on a symbolic configuration the current path condition is enriched with the rule's condition.

Example 4. The last rule for `if` from the (already transformed) IMP semantics is further transformed into the following rule in $\mathcal{S}^{\mathfrak{s}}$:

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

4.4 Defining the Symbolic Transition System

The triple $(\Sigma^{\mathfrak{s}},\ \mathcal{T}^{\mathfrak{s}},\ \mathcal{D}^{\mathfrak{s}})$ defines a language $\mathcal{L}^{\mathfrak{s}}$. Then, the transition system $(\mathcal{T}^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}, \Rightarrow_{\mathcal{S}^{\mathfrak{s}}})$ can be defined using Definitions 1 and 2 applied to $\mathcal{L}^{\mathfrak{s}}$. For this, we note that both sides of the rules of the form (2) are terms in $T_{\Sigma^{\mathfrak{s}}, \mathit{Cfg}^{\mathfrak{s}}}(\mathit{Var})$, thus, according to Definition 1 applied to $\mathcal{L}^{\mathfrak{s}}$, they are (basic) patterns of $\mathcal{L}^{\mathfrak{s}}$, and then Definition 2 for $\mathcal{L}^{\mathfrak{s}}$ gives us the transition system $(\mathcal{T}^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}, \Rightarrow_{\mathcal{S}^{\mathfrak{s}}})$.

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

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

The next lemma shows that the symbolic transition system forward-simulates the concrete transition system. We denote by $\alpha^{\mathfrak{s}} \in \mathcal{S}^{\mathfrak{s}}$ the rule obtained by transforming $\alpha \in \mathcal{S}$ (Section 4.3).

Lemma 2. *$(\mathcal{T}^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}, \Rightarrow_{\mathcal{S}^{\mathfrak{s}}})$ forward simulates $(\mathcal{T}_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}})$: for all configurations γ, symbolic configurations $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$ and rules $\alpha \in \mathcal{S}$, if $\gamma \models \langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$ and $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ then there exists $\langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$ such that $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle \overset{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$ and $\gamma' \models \langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$.*

Proof. From $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ we obtain $\alpha \triangleq (l \wedge b \Rightarrow r) \in \mathcal{S}$ and $\rho : \mathit{Var} \to \mathcal{T}$ such that $\gamma = \rho(l)$, $\rho \models b$, and $\gamma' = \rho(r)$. Recall that $\alpha^{\mathfrak{s}} \triangleq (\langle l, \psi \rangle \Rightarrow \langle r, \psi \wedge b \rangle)$.

From $\gamma \models \langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$ we obtain $\vartheta : V^{\mathfrak{s}} \to \mathcal{D}$ such that $\gamma = \vartheta(\pi^{\mathfrak{s}})$ and $\vartheta \models \phi^{\mathfrak{s}}$.

Using Lemma 1 we obtain the valuation σ such that $\sigma(l) = \pi^{\mathfrak{s}}$ and $\rho(x) = \vartheta(\sigma(x))$ for each $x \in \mathit{Var}$.

We define $\pi'^{\mathfrak{s}} \triangleq \sigma(r)$ and $\phi'^{\mathfrak{s}} \triangleq \sigma(b) \wedge \phi^{\mathfrak{s}}$. Consider the valuation $\sigma[\psi \mapsto \phi^{\mathfrak{s}}]$, which behaves like σ on $\mathit{Var} \setminus \{\psi\}$ and maps ψ to $\phi^{\mathfrak{s}}$.

We prove $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle \overset{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$ using the valuation $\sigma[\psi \mapsto \phi^{\mathfrak{s}}]$.

- First, $(\sigma[\psi \mapsto \phi^{\mathfrak{s}}])(\langle l, \psi \rangle) = \langle \sigma(l), \phi^{\mathfrak{s}} \rangle = \langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$, since ψ does not occur in the rule, thus, the left-hand side $\langle l, \psi \rangle$ of the rule $\alpha^{\mathfrak{s}}$ matches $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$.
- Second, $\langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle = \langle \sigma(r), \sigma(b) \wedge \phi^{\mathfrak{s}} \rangle = \langle (\sigma[\psi \mapsto \phi^{\mathfrak{s}}])(r), (\sigma[\psi \mapsto \phi^{\mathfrak{s}}])(\psi \wedge b) \rangle = (\sigma[\psi \mapsto \phi^{\mathfrak{s}}])](\langle r, \psi \wedge b \rangle)$. Thus, $\alpha^{\mathfrak{s}}$ rewrites $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$ to $\langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$.

This proves $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle \overset{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$. There remains to prove $\gamma' \models \langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$.

For this we use the same valuation $\vartheta : V^\mathfrak{s} \to \mathcal{D}$ as above. We have $\vartheta(\pi'^\mathfrak{s}) = \vartheta(\sigma(r))$, which, using Lemma 1, is $\rho(r)$, and the latter equals γ', cf. beginning of the proof. Thus, $\gamma' = \vartheta(\pi'^\mathfrak{s})$.

On the other hand, $\vartheta(\phi'^\mathfrak{s}) = \vartheta(\sigma(b) \wedge \phi^\mathfrak{s}) = \vartheta(\sigma(b)) \wedge \vartheta(\phi^\mathfrak{s}) = \rho(b) \wedge \vartheta(\phi^\mathfrak{s})$. We have:

- $\rho(b) = \mathit{true}$ because we have $\rho \models b$ from the beginning of the proof;
- $\vartheta(\phi^\mathfrak{s}) = \mathit{true}$ because $\vartheta \models \phi^\mathfrak{s}$, also from the beginning of the proof;

which implies $\rho(b) \wedge \vartheta(\phi^\mathfrak{s}) = \mathit{true}$, thus, $\vartheta(\phi'^\mathfrak{s}) = \mathit{true}$, which together with $\gamma' = \vartheta(\pi'^\mathfrak{s})$ proved above implies $\gamma' \models \langle \pi'^\mathfrak{s}, \phi'^\mathfrak{s} \rangle$, which completes the proof. □

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

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

Theorem 1 (Coverage). *If $\gamma \overset{\beta}{\Longrightarrow}_\mathcal{S} \gamma'$ and $\gamma \models \langle \pi^\mathfrak{s}, \phi^\mathfrak{s} \rangle$ then there is a symbolic configuration $\langle \pi'^\mathfrak{s}, \phi'^\mathfrak{s} \rangle$ such that $\gamma' \models \langle \pi'^\mathfrak{s}, \phi'^\mathfrak{s} \rangle$ and $\langle \pi^\mathfrak{s}, \phi^\mathfrak{s} \rangle \overset{\beta^\mathfrak{s}}{\Longrightarrow}_{\mathcal{S}^\mathfrak{s}} \langle \pi'^\mathfrak{s}, \phi'^\mathfrak{s} \rangle$*

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

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

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

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

Proof. The transition $\langle \pi^\mathfrak{s}, \phi^\mathfrak{s} \rangle \overset{\alpha^\mathfrak{s}}{\Longrightarrow}_{\mathcal{S}^\mathfrak{s}} \langle \pi'^\mathfrak{s}, \phi'^\mathfrak{s} \rangle$ is obtained by applying a symbolic rule $\alpha^\mathfrak{s} \triangleq (\langle l, \psi \rangle \Rightarrow \langle r, \psi \wedge b \rangle) \in \mathcal{S}^\mathfrak{s}$, with some valuation that has the form $(\sigma[\psi \mapsto \phi^\mathfrak{s}]) : \mathit{Var} \to \mathcal{T}^\mathfrak{s}|_\Sigma$. Thus, $\sigma(l) = \pi^\mathfrak{s}$, $\pi'^\mathfrak{s} = \sigma(r)$, and $\phi'^\mathfrak{s} = \phi^\mathfrak{s} \wedge \sigma(b)$.

From $\gamma' \models \langle \pi'^\mathfrak{s}, \phi'^\mathfrak{s} \rangle$ we obtain $\vartheta : V^\mathfrak{s} \to \mathcal{T}$ such that $\gamma' = \vartheta^\mathfrak{s}(\pi'^\mathfrak{s}) = \vartheta^\mathfrak{s}(\sigma(r)) = (\vartheta^\mathfrak{s} \circ \sigma)(r)$ and $\mathit{true} = \vartheta^\mathfrak{s}(\phi'^\mathfrak{s}) = \vartheta^\mathfrak{s}(\phi^\mathfrak{s}) \wedge (\vartheta^\mathfrak{s} \circ \sigma)(b)$, thus, $\vartheta^\mathfrak{s}(\phi^\mathfrak{s}) = \mathit{true}$ and $(\vartheta^\mathfrak{s} \circ \sigma)(b) = \mathit{true}$.

Consider also $\rho : \mathit{Var} \to \mathcal{T} \triangleq \vartheta^\mathfrak{s} \circ \sigma$, and let $\gamma \triangleq \rho(l)$. We have:

- on the one hand, $\gamma = \rho(l) = (\vartheta^\mathfrak{s} \circ \sigma)(l) = \vartheta^\mathfrak{s}(\sigma(l)) = \vartheta^\mathfrak{s}(\pi^\mathfrak{s})$, i.e., $\gamma = \vartheta^\mathfrak{s}(\pi^\mathfrak{s})$;

– on the other hand, $\vartheta^{\mathfrak{s}}(\phi^{\mathfrak{s}}) = \mathit{true}$ was obtained above;

which proves $\gamma \models \langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$. There remains to prove $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$. To prove this we consider the rule $\alpha = (l \wedge r \Rightarrow b) \in \mathcal{S}$ whose symbolic version is $\alpha^{\mathfrak{s}}$ from the beginning of the proof, and the valuation $\rho = \vartheta^{\mathfrak{s}} \circ \sigma$ from above. We have:

– $\gamma = \rho(l)$ by definition of γ;
– $\rho(b) = \mathit{true}$, which is just $(\vartheta^{\mathfrak{s}} \circ \sigma)(b) = \mathit{true}$ that we obtained above;
– $\gamma' = \rho(r)$, since we obtained above $\gamma' = (\vartheta^{\mathfrak{s}} \circ \sigma)(r)$.

This proves $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ and completes the proof. □

A consequence of this lemma is the *precision* theorem; it says that if a sequence $\beta^{\mathfrak{s}}$ of symbolic rules can be executed starting in some initial symbolic configuration and reaches a satisfiable final symbolic configuration (thus, implicitly, all intermediary path conditions are satisifiable, since the final path condition is logically stronger than all the intermediary ones) then the corresponding sequence of concrete rules can be fired as well.

Theorem 2 (Precision). *If* $\langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle \overset{\beta^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$ *and* $\gamma' \models \langle \pi'^{\mathfrak{s}}, \phi'^{\mathfrak{s}} \rangle$ *then there exists a configuration* γ *such that* $\gamma \models \langle \pi^{\mathfrak{s}}, \phi^{\mathfrak{s}} \rangle$ *and* $\gamma \overset{\beta}{\Longrightarrow}_{\mathcal{S}} \gamma'$.

6 Implementation

In this section we present a prototype tool implementing our symbolic execution approach. In Section 6.1 we briefly present our tool and its integration within the $\mathbb{K}$ framework. In Section 6.2 we illustrate the most significant features of the tool by the means of use cases involving nontrivial languages and programs.

6.1 Symbolic Execution within the $\mathbb{K}$ Framework

Our tool is part of $\mathbb{K}$ [20,24], a semantic framework for defining operational semantics of programming languages. In $\mathbb{K}$ the definition of a language, say, $\mathcal{L}$, is compiled into a 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 4.3. The effect is that the compiled rewrite theory defines 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 current version of the tool provides symbolic support for some of the most standard $\mathbb{K}$ data types: Booleans, integers, strings, as well as arrays whose size, index, and content 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 the axioms of the symbolic data domains as simplification rules and, possibly, calls to the Z3 SMT solver[6]. For efficiency reasons the SMT solver is called only if the rules add non-trivial formula to path conditions, which cannot be simplified to *true* or *false* by the axioms of the symbolic domains. Users can also fine-tune the amount of calls to the solver in order to achieve a balance between the precision and the execution time of their symbolic execution. There is also an option for displaying the transformed $\mathbb{K}$ definitions.

The current version of the tool has some limitations, which we are planning to deal with in the future: only data constants, not full data subterms, are replaced with variables, the tool is connected to only one prover (Z3), and it provides only a limited support for building applications based on symbolic execution.

6.2 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 [10]. 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. Our tool is applied on the current definitions of SIMPLE and KOOL.

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 telling it to generate the symbolic semantics of the language by applying the transformations described in Section 4.3.

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

Fig. 6. `sum.imp`

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

Fig. 7. `log.imp`

Programs such as `sum.imp` shown in Figure 6 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 ∧ (n − 1 > 0) ∧ ¬((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 [16]. 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 (automatic) extension of the syntax and semantics of a language for including labels that are used as atomic propositions in the LTL formulas. Predicates on the program's variables can be used as propositions in the formulas as well, using the approach outlined in [15].

Consider for instance the program `log.imp` in Figure 7, which computes the integer binary logarithm of an integer read from the input. We prove that whenever the loop visits the label `L`, the inequalities $\mathtt{x} * 2^{\mathtt{k}} \leq \mathtt{a} < (\mathtt{x}+1) * 2^{\mathtt{k}}$ hold. The invariant was guessed using several step-wise executions. 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.

SIMPLE, Symbolic Arrays, and Bounded Model Checking. We illustrate symbolic arrays in the SIMPLE language and shows how the $\mathbb{K}$ runner can directly

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

```
void main() {
     int n = read();
     int j = read();
     int x = read();
     int a[n], i = 0;
     while (i < n) {
       a[i] = read();
       i = i + 1;
     }
     init(a, x, j);
}
```

Fig. 8. SIMPLE program: `init-arrays`

be used for performing bounded model checking. In the program in Figure 8, 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:

```
$ 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.

KOOL: Testing Virtual Method Calls on Lists. Our last example (Figure 9) 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 9 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

```
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");
    }
  }
}
```

Fig. 9. `lists.kool`: implementation of lists in KOOL

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>
   e1 = x ∧Bool ¬Bool (e1 = e2)
</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.

6.3 The Implementation of the Tool

Our tool was developed as an extension of the $\mathbb{K}$ 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 4.3). These transformations are inserted when the $\mathbb{K}$ compiler is called with the `-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 4.3 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. For this reason, the transformer that collects the path condition also adds, as a side condition to rewrite 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. A 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`.

7 Conclusion and Future Work

We have presented a formal and generic framework for the symbolic execution of programs in languages having operational semantics defined by term-rewriting. Starting from the formal definition of a language $\mathcal{L}$, the symbolic version $\mathcal{L}^{\mathfrak{s}}$ of the language is automatically constructed, by extending the datatypes used in $\mathcal{L}$ with symbolic values, and by modifying the semantical rules of $\mathcal{L}$ in order to make them process symbolic values appropriately. The symbolic semantics of $\mathcal{L}$ is then the (usual) semantics of $\mathcal{L}^{\mathfrak{s}}$, and symbolic execution of programs in $\mathcal{L}$ is the (usual) execution of the corresponding programs in $\mathcal{L}^{\mathfrak{s}}$, which is the application of the rewrite rules of the semantics of $\mathcal{L}^{\mathfrak{s}}$ to programs. Our symbolic execution has the natural properties of *coverage*, meaning that to each concrete execution there is a feasible symbolic one on the same path of instructions, and *precision*, meaning that each feasible symbolic execution has a concrete execution on the same path. These results were obtained by carefully constructing definitions about the essentials of programming languages, in an algebraic and term-rewriting setting. We have implemented a prototype tool in the $\mathbb{K}$ framework and have illustrated it by instantiating it to several languages defined in $\mathbb{K}$.

Future Work. We are planning to use symbolic execution as the basic mechanism for the deductive systems for program logics also developed in the $\mathbb{K}$ framework (such as reachability logic [21] and our own circular equivalence logic [14]). More generally, our symbolic execution can be used for program testing, debugging, and verification, following the ideas presented in related work, but with the added value of being language independent and grounded in formal operational semantics. In order to achieve that, we have to develop a rich domain of symbolic values, able to handle e.g., heaps, stacks, and other common data types.

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`). We would like to thank the reviewers for their helpful comments. The work presented here was supported in part by Contract 161/15.06.2010, SMIS-CSNR 602-12516 (DAK).

References

1. Armando, A., Benerecetti, M., Mantovani, J.: Model checking linear programs with arrays. In: Proceedings of the Workshop on Software Model Checking, vol. 144-3, pp. 79–94 (2006)
2. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, New York (1998)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)

4. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) ACM Conference on Computer and Communications Security, pp. 322–335. ACM (2006)
5. de Halleux, J., Tillmann, N.: Parameterized unit testing with pex. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 171–181. Springer, Heidelberg (2008)
6. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Dillon, L.K.: Verifying general safety properties of Ada tasking programs. IEEE Trans. Softw. Eng. 16(1), 51–63 (1990)
8. Escobar, S., Meseguer, J., Sasse, R.: Variant narrowing and equational unification. Electr. Notes Theor. Comput. Sci. 238(3), 103–119 (2009)
9. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223. ACM (2005)
10. Hills, M., Roşu, G.: KOOL: An application of rewriting logic to language prototyping and analysis. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 246–256. Springer, Heidelberg (2007)
11. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
12. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
13. Li, G., Ghosh, I., Rajan, S.P.: KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 609–615. Springer, Heidelberg (2011)
14. Lucanu, D., Rusu, V.: Program equivalence by circular reasoning. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 362–377. Springer, Heidelberg (2013)
15. Lucanu, D., Şerbănuţă, T.F., Roşu, G.: $\mathbb{K}$ Framework Distilled. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 31–53. Springer, Heidelberg (2012)
16. Meseguer, J.: Rewriting logic and Maude: Concepts and applications. In L. Bachmair, editor, *RTA*. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 1–26. Springer, Heidelberg (2000)
17. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. Higher-Order and Symbolic Computation 20(1-2), 123–160 (2007)
18. Păsăreanu, C.S., Visser, W.: Verification of Java Programs Using Symbolic Execution and Invariant Generation. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 164–181. Springer, Heidelberg (2004)
19. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT 11(4), 339–353 (2009)
20. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
21. Roşu, G., Ştefănescu, A.: Checking reachability using matching logic. In: Leavens, G.T., Dwyer, M.B. (eds.) OOPSLA, pp. 555–574. ACM (2012)
22. Schmitt, P.H., Weiß, B.: Inferring invariants by symbolic execution. In: Proceedings of 4th International Verification Workshop, VERIFY 2007 (2007)

23. Sen, K., Marinov, D., Agha, G.: 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, pp. 263–272. ACM (2005)
24. Serbanuta, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Rosu, G.: The K primer (version 2.5). In: Hills, M. (ed.) K 2011. Electronic Notes in Theoretical Computer Science (2011) (to appear)
25. Şerbănuţă, T.-F., Roşu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Inf. Comput. 207(2), 305–340 (2009)
26. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In: ISSTA, pp. 157–168. ACM (2006)
27. Staats, M., Păsăreanu, C.S.: Parallel symbolic execution for structural test generation. In: Tonella, P., Orso, A. (eds.) ISSTA, pp. 183–194. ACM (2010)
28. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Avrunin, G.S., Rothermel, G. (eds.) ISSTA, pp. 97–107. ACM (2004)