A Generic Approach to Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

A Generic Framework for Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

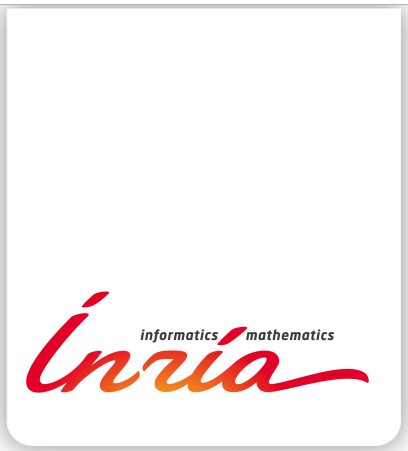
RESEARCH REPORT

N° 8189

December 2012

Project-Team Dart

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



A Generic Framework for Symbolic Execution

Andrei Arusoaie*, Dorel Lucanu†, Vlad Rusu‡

Project-Team Dart

Research Report n° 8189 — December 2012 — 25 pages

Abstract: We propose a language-independent symbolic execution framework. The approach is parameterised by a language definition, which consists of a signature for the language's syntax and execution infrastructure, a model interpreting the signature, and rewrite rules for the language's operational semantics. Then, symbolic execution amounts to performing a so-called symbolic rewriting, which consists in changing both the model and the manner in which the operational semantics rules are applied. 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.

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

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

RESEARCH CENTRE
LILLE – NORD EUROPE

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

Un cadre général pour l'exécution symbolique

Résumé : Nous proposons un cadre général pour l'exécution symbolique de programmes, qui est indépendant des langages dans lesquels les programmes en question sont écrits. L'approche est paramétrisée par une définition de langaage, qui consiste en une signature pour la syntaxe du langage et pour son infrastructure, un modèle interprétant la signature, et un ensemble de règles de réécriture définissant la sémantique opérationnelle du langage. L'exécution symbolique revient alors à modifier le modèle et la façon d'appliquer les règles sémantiques sur celui-ci. Nous démontrons que l'exécution symbolique possède les propriétés attendues par rapport à l'exécution concrète. Nous avons implémenté notre approche dans un outil prototype dans la K framework. L'aspect générique de l'outil est mis en évidence par son instanciation sur plusieurs langages. Nous montrons enfin comment l'outil permet l'exécution symbolique et le model checking de programmes.

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

1 Introduction

Symbolic execution is a well-known program analysis technique introduced in 1976 by James C. King [13]. 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 [17]. 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 [14], [26], invariant detection [16], model checking [12], and proving program correctness [25, 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 the control flow of a program is determined by symbolic values (e.g., the next instruction to be executed is a conditional statement, whose condition depends on symbolic values) then there is a branching in the tree. The path condition is then 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 symbolic execution as the application of rewrite rules in the semantics by *symbolic rewriting*, which consists of applying rules with *unification* instead of matching, on a model of *patterns* instead of ground terms. 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. Since $\mathbb{K}$ is based on standard rewriting (with matching, not with unification) we first show how symbolic rewriting can be emulated by applying certain modified rewrite rules (obtained by automatically transforming the original ones) with matching. This relates the present formalisation of symbolic execution with an earlier approach we developed in [2]. We briefly describe our implementation as a language-engineering tool, and demonstrate its genericity by instantiating it on several nontrivial languages defined in $\mathbb{K}$. The examples illustrate program execution as well as Linear Temporal Logic model checking and bounded model checking using our tool.

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

[1] Most existing operational semantics styles (small-step, big-step, reduction with evaluation contexts, ...) have been shown to be representable in this way in [24].

and are based on informal semantics. Here we mention those most strongly related to ours.

Java PathFinder [18] 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 [10], 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 [13]. It can also be used for proving program correctness. There are several tools (e.g. Smallfoot [3, 27]) which use symbolic execution together with separation logic to prove Hoare triples. There are also approaches that attempt to automatically detect invariants in programs([16], [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 [15]. Such analyses rely on powerful unification-modulo-theories algorithms [9], 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 some background material used in the rest of the paper: our running example (the simple imperative language IMP) and its definition in the $\mathbb{K}$ framework, and a generic framework for language definitions as triples consisting of a first-order signature, a model for the signature, an rewrite rules operating on the model. This makes 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 3 shows how symbolic execution can be defined by symbolic rewriting, which consists in applying the language's semantical rules with unification instead of matching, on a model consisting of patterns instead of ground terms. We also establish the coverage and precision results stated in

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

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

this introduction. Section 4 describes an implementation of our approach in the $\mathbb{K}$ framework. Since $\mathbb{K}$ is based on standard rewriting on ground terms, we first show how our symbolic rewriting can be encoded by standard rewriting. We then instantiate our verifier to nontrivial languages defined in $\mathbb{K}$ and demonstrate it on examples. We conclude and present future work in Section 5.

2 Background

2.1 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.

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 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. Fig. 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

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

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

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.

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:

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

Here, BE ranges over $BExp \setminus \{\mathit{false}, \mathit{true}\}$, B ranges over the 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.

2.2 Language Definitions

In this section we identify the ingredients of language definitions in a 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.

Signature: Φ is a many-sorted first-order signature. It consists of a many-sorted algebraic signature Σ containing function symbols, and a set Π of predicate symbols. Σ includes at least a sort *Cfg* for *configurations* as well as sorts for the syntax of the language $\mathcal{L}$, e.g., expressions and statements. Σ may also include other data sorts, depending on the datatypes occuring in the language $\mathcal{L}$ (e.g., Booleans, 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 set of variables V, let $T_\Sigma(V)$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(V)$ denote the set of terms of sort s with variables, and $\mathit{var}(t)$ denote the set of variables occurring in the term t.

Model: $\mathcal{T}$ is a Φ-model, i.e., it interprets every function and predicate in Φ. We assume that it interprets the data sorts and their operations according to a given Σ^{Data}-model $\mathcal{D}$. For simplicity, we write in the sequel $true, false, 0, 1, \ldots$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}, \mathcal{D}_0, \mathcal{D}_1$, etc. $\mathcal{T}$ interprets the non-data sorts as the free Σ-model generated by $\mathcal{D}$, i.e., as ground terms over the signature $(\Sigma \setminus \Sigma^{\mathsf{Data}}) \cup \mathcal{D}$. We denote by $\rho \models \phi$ the satisfaction of a Φ-formulas ϕ by a valuation $\rho : Var \to \mathcal{T}$.

Rules: $\mathcal{S}$ is a set of semantical rules, of the form $\varphi \Rightarrow \varphi'$ where φ, φ' are *elementary patterns.*

Definition 1 ([19]) *An* elementary pattern *over a set of variables Var is an expression of the form $\pi \wedge \phi$, where $\pi \in T_{\Sigma, Cfg}(Var)$ is a* basic pattern *and ϕ is a Φ-formula called the pattern's* condition*. If $\gamma \in T_{Cfg}$ and $\rho : Var \to \mathcal{T}$ we write $(\gamma, \rho) \models \pi \wedge \phi$ for $\gamma = \pi\rho$ and $\rho \models \phi$. We let $[\![\varphi]\!]$ denote the set $\{\gamma \in \mathcal{T}_{Cfg} \mid$ there is $\rho : Var \to \mathcal{T}$ s.t. $(\gamma, \rho) \models \varphi\}$.*

The above definition is a particular case of a definition in [19]. There, a pattern is a first-order logic formula with configuration basic-patterns as sub-formulas. A basic pattern π defines a set of (concrete) configurations, and the condition ϕ gives additional constraints these configurations must satisfy. We identify basic patterns π with elementary patterns $\pi \wedge true$. Sample patterns are $\langle\langle I_1 \curvearrowright C\rangle_{\mathsf{k}} \langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \curvearrowright C\rangle_{\mathsf{k}} \langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$.

Definition 2 (rule, transition system) *A* rule *is a pair of elementary patterns over a set of variables Var, of the form $\varphi \Rightarrow \varphi'$. Any set $\mathcal{S}$ of rules defines a transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ iff there exist $\alpha \triangleq (\varphi \Rightarrow \varphi') \in \mathcal{S}$ and $\rho : Var \to \mathcal{T}$ satisfying $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$.*

Remark 1 *Any elementary pattern $\pi \wedge \phi$ can be transformed into $\pi' \wedge \phi'$ such that $[\![\pi \wedge \phi]\!] = [\![\pi \wedge \phi]\!]$, π' is linear, and all its data subterms are variables. This is required for applying semantical rules with unification (cf. Section 3.1) To perform this transformation, it is enough to replace all duplicated variables and all non-variable data subterms of π by fresh variables, and to add constraints in ϕ' that equate the fresh variables to the subterms they replaced.*

Example For configurations containing a k cell and an env cell with valuations, the elementary pattern $\langle\langle X \curvearrowright Y\rangle_{\mathsf{k}} \langle Y \mapsto A +_{Int} 1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A \neq_{Int} -1$ with X, Y variables of sort *Id* and A of sort *Int* is nonlinear because Y occurs twice. It contains the non-variable data terms $A +_{Int} 1$. It is transformed into $\langle\langle X \curvearrowright Y\rangle_{\mathsf{k}} \langle Y' \mapsto A'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge Y' =_{Id} Y \wedge_{Bool} A' =_{Int} A +_{Int} 1 \wedge_{Bool} A \neq_{Int} -1$. □

We show how the definition of IMP fits the theoretical language-definition framework given abobe. Nonterminals from the syntax (*Int*, *Bool*, *AExp*, ...) 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 $\_\texttt{+}\_ : AExp \times AExp \to AExp$. 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 $\langle\langle I_1 \curvearrowright C\rangle_{\mathsf{k}} \langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$ is an elementary pattern in which $=_{Int}$ is a predicate symbol, I_1, I_2 are 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 multiset of maps $X \mapsto I$, where X ranges over identifiers *Id* and I over the integers. Predicate symbols such as $=_{Int}, \leq_{Int}$ are interpreted by the corresponding predicates over 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 in which data subterms are replaced by their interpretations, e.g., `if` $1 >_{Int} 0$ `then {} else {}` is intepreted as `if` $\mathcal{D}_{true}$ `then {} else {}`.

3 Symbolic Execution

In this section we present a symbolic execution approach for languages defined using the language-definition framework presented in the previous section. We prove that the transition system generated by symbolic execution forward-simulates the one generated by concrete execution, and that the transition system generated by concrete execution backward-simulates the one generated by symbolic execution (restricted to satisfiable patterns). These properties, which we called coverage and precision, are the most naturally expected ones from a symbolic execution framework. They allow to perform analyses on symbolic programs, and to soundly transfer the results of those analyses to concrete instances of the symbolic programs in question.

3.1 Unification

We shall be using unification for symbolic execution. In this section we define unification and prove a technical lemma used later in the paper.

Definition 3 (Unifiers) *A* symbolic unifier *of two terms t_1, t_2 is any substitution $\sigma : var(t_1) \uplus var(t_2) \to T_\Sigma(Z)$ for some set Z of variables such that $t_1\sigma = t_2\sigma$. A* concrete unifier *of terms t_1, t_2 is any valuation $\rho : var(t_1) \uplus var(t_2) \to \mathcal{T}$ such that $t_1\rho = t_2\rho$. A symbolic unifier σ of two terms t_1, t_2 is a* most general unifier of t_1, t_2 with respect to concrete unification *whenever, for all concrete unifiers ρ of t_1 and t_2, there is a valuation η such that $\sigma\eta = \rho$.*

We call a symbolic unifier satisfying the above a *most general unifier* (even though the standard notion of most general unifier in algebraic specifications and rewriting is a different one.)

We say that terms t_1, t_2 are symbolically (resp. concretely) unifiable if they have a symbolic (resp. concrete) unifier. The next lemma gives conditions under which concretely unifiable terms are symbolically unifiable.

Lemma 1 *All linear, concretely unifiable terms $t_1, t_2 \in T_\Sigma(Var)$ of non-data sorts, such that all their data subterms are variables, are symbolically unifiable by a most general unifier $\sigma^{t_1}_{t_2} : var(t_1) \uplus var(t_2) \to T_\Sigma(var(t_1) \uplus var(t_2))$.*

Proof By induction on the structure of, say, t_1. In the base case, $t_1 \in Var$, and we take $\sigma^{t_1}_{t_2} \triangleq (t_1 \mapsto t_2) \uplus id|_{var(t_2)}$, i.e., $\sigma^{t_1}_{t_2}$ maps t_1 to t_2, and is the identity on $var(t_2)$. Obviously, $\sigma^{t_1}_{t_2}$ is a unifier of t_1, t_2, since $t_1\sigma^{t_1}_{t_2} = t_2 = t_2\sigma^{t_1}_{t_2}$. To show that $\sigma^{t_1}_{t_2}$ is most general, consider any concrete unifier of t_1, t_2, say, ρ. Then, $t_1\sigma^{t_1}_{t_2}\rho = t_2\rho$ because $\sigma^{t_1}_{t_2}$ maps t_1 to t_2, and $t_2\rho = t_1\rho$ because ρ is a concrete unifier. Thus, $t_1\sigma^{t_1}_{t_2}\rho = t_1\rho$. Moreover, for all $x \in var(t_2)$, $x\sigma^{t_1}_{t_2}\rho = x\rho$ since $\sigma^{t_1}_{t_2}$ is the identity on $var(t_2)$. Thus, for all $y \in var(t_1) \uplus var(t_2) (= \{t_1\} \uplus var(t_2))$, $y\sigma^{t_1}_{t_2}\rho = y\rho$, which proves the fact that $\sigma^{t_1}_{t_2}$ is a most general unifier (by taking $\eta = \rho$ in Definition 3 of unifiers). The fact that the codomain of $\sigma^{t_1}_{t_2}$ is $T_\Sigma(var(t_1) \uplus var(t_2))$ results from its construction.

In the inductive step, $t_1 = f(s_1, \ldots, s_n)$ with $f \in \Sigma \setminus \Sigma^{Data}$ [2] $n \geq 0$, and $s_1, \ldots, s_n \in T_\Sigma(Var)$. For t_2 there are two subcases:

[2] $f \in \Sigma \setminus \Sigma^{Data}$ because the contrary implies that t_1 has a sort *Data*, contradicting the lemma's hypotheses.

- t_2 is a variable. Then, let $\sigma_{t_2}^{t_1} \triangleq (t_2 \mapsto t_1) \uplus id|_{var(t_1)}$, i.e., $\sigma_{t_2}^{t_1}$ maps t_2 to t_1, and is the identity on $var(t_1)$. We prove that $\sigma_{t_2}^{t_1}$ is a most general unifier with codomain $T_\Sigma(var(t_1) \uplus var(t_2))$ like in the base case.

- $t_2 = g(u_1, \ldots, u_m)$ with $g \in \Sigma$, $m \geq 0$, and $u_1, \ldots, u_m \in T_\Sigma(Var)$. Let ρ be a concrete unifier of t_1, t_2, thus, $(f\rho)(s_1\rho \ldots s_n\rho) =_{\mathcal{T}} (g\rho)(u_1\rho \ldots u_m\rho)$, where we emphasize by subscripting the equality symbol with $\mathcal{T}$ that the equality is that of the model $\mathcal{T}$. Since $\mathcal{T}$ interprets non-data terms syntactically, we have $f\rho = f$, which implies $f = g$, $g\rho = g$, $m = n$, and $s_i\rho = u_i\rho$ for $i = 1, \ldots, n$. Since t_1 and t_2 are linear and all their data subterms are variables, the subterms s_i and u_i also have these properties. Using the induction hypothesis we build most-general-unifiers $\sigma_{u_i}^{s_i}$ of s_i and u_i, which have codomains $T_\Sigma(var(s_i) \uplus var(u_i))$, for $i = 1, \ldots, n$. Let then $\sigma_{t_2}^{t_1} \triangleq \biguplus_{i=1}^{n} \sigma_{u_i}^{s_i}$.

  First, $\sigma_{t_2}^{t_1}$ is a substitution of $var(t_1) \uplus var(t_2)$ into $T_\Sigma(var(t_1) \uplus var(t_2))$ since $var(t_1) = \uplus_{i=1}^{n} var(s_i)$ and $var(t_2) = \uplus_{i=1}^{n} var(u_i)$. Note that these equalities hold thanks to the linearity of t_1, t_2. Second, $\sigma_{t_2}^{t_1}$ is a unifier of t_1, t_2 since all $\sigma_{u_i}^{s_i}$ are so. Third, we prove that $\sigma_{t_2}^{t_1}$ is a most general unifier of t_1, t_2. Consider any concrete unifier ρ of t_1 and t_2, thus, $s_i\rho = u_i\rho$ for $i = 1, \ldots, n$. From the fact that all the $\sigma_{u_i}^{s_i}$ are most-general-unifiers of s_i and u_i for $i = 1, \ldots, n$, we obtain the existence of valuations η_i such that $\sigma_{u_i}^{s_i}\eta_i = \rho|_{(var(s_i) \uplus var(u_i))}$, for $i = 1, \ldots, n$. Then, $\eta \triangleq \biguplus_{i=1}^{n} \eta_i$, which is also well-defined thanks to the linearity of t_1 and t_2, has the property that $\sigma_{t_2}^{t_1}\eta = \rho$, which proves that $\sigma_{t_2}^{t_1}$ is a most general unifier of t_1 and t_2 and concludes the proof. □

Remark 2 *If the conditions of Lemma 1 are replaced by "t_1 and t_2 are concretely unifiable and have non-data sorts t_2 is linear, and all the elements of $var(t_2)$ have a data sort" then the substitution $\sigma_{t_2}^{t_1}$ constructed in the proof of Lemma 1 can be decomposed as $\mu_{t_2}^{t_1} \uplus id_{var(t_2)}$ where $\mu_{t_2}^{t_1} : var(t_1) \to T_\Sigma(var(t_2))$ is a match of t_1 on t_2, i.e., $t_1\mu_{t_2}^{t_1} = t_2$. This is proved by noting that the first case (t_2 is a variable) in the inductive step of the proof of Lemma 1 is impossible (t_2 should be of a data sort because it is a variable, and of a non-data sort by our new hypotheses). The rest of the proof builds $\sigma_{t_2}^{t_1}$ as the identity on $var(t_2)$ and as a match of t_1 on t_2. This observation is important because we shall need to implement unification by matching. Moreover, if there is a substitution $\mu_{t_2}^{t_1} : var(t_1) \to T_\Sigma(var(t_2))$ such that $t_1\mu_{t_2}^{t_1} = t_2$, then this substitution is unique since matching of non-data terms is syntactical. Together with the above observations this implies that $\mu_{t_2}^{t_1} \uplus id_{var(t_2)}$ coincides with the most-general unifier $\sigma_{t_2}^{t_1}$ from the proof of Lemma 1.*

3.2 Symbolic Transition Relation

Symbolic execution generates a *symbolic transition system* whose states are equivalence of elementary patterns (modulo logical equivalence of their condition), and whose transition relation is obtained by applying rewrite rules with the most-general unifiers whose construction is given by Lemma 1.

We first define the relation $\sim$ between pattern by $\pi \wedge \phi \sim \pi' \wedge \phi'$ iff $\pi = \pi'$ and $\models \phi \leftrightarrow \phi'$, i.e., the basic patterns π and π' are syntactically equal terms, and the equivalence $\phi \leftrightarrow \phi'$ of the conditions ϕ and ϕ' is logically valid. The relation $\sim$ is an equivalence. We let $[\varphi]_\sim$ denote the equivalence class of φ.

Definition 4 (Symbolic transition relation) *We define the symbolic transition relation $\Rightarrow_{\mathcal{S}}^{\mathfrak{s}}$ by: $[\varphi]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi']_\sim$ iff $\varphi \triangleq \pi \wedge \phi$, there is a rule $\alpha \triangleq \varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ with $\varphi_i \triangleq \pi_i \wedge \phi_i$ for $i = 1, 2$, and $var(\pi) \cap var(\pi_1) = \emptyset$ such that π, π_1 are concretely unifiable, and $\varphi' = \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, where $\sigma_\pi^{\pi_1}$ is the most general symbolic unifier of π, π_1 (cf. proof of Lemma 1).*

In the rest of the paper, for patterns $\varphi \triangleq \pi \wedge \phi$ we let $var(\varphi) \triangleq var(\pi, \phi)$, and for rules $\alpha \triangleq \varphi_1 \Rightarrow \varphi_2$ we let $var(\alpha) \triangleq var(\varphi_1, \varphi_2)$. Moreover, for symbolic transitions $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ we assume without restriction on generality that $var(\varphi) \cap var(\alpha) = \emptyset$, which can always be obtained by variable renaming.

3.3 Coverage

Lemma 2 *If $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\varphi]\!]$ then there exists φ' such that $\gamma' \in [\![\varphi']\!]$ and $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$.*

Proof Let $\varphi \triangleq \pi \wedge \phi$. From $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ we obtain the rule $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ and the valuation $\rho : Var \to \mathcal{T}$ such that $\gamma = \pi_1\rho$, $\rho \models \phi_1$, $\rho \models \phi_2$, and $\gamma' = \pi_2\rho$. From $\gamma \in [\![\varphi]\!]$ we obtain the valuation $\mu : Var \to \mathcal{T}$ such that $\mu\gamma = \pi\mu$ and $\mu \models \phi$. Thus, π_1 and π are concretely unifiable (by their concrete unifier $\rho|_{var(\pi_1)} \uplus \mu|_{var(\pi)}$). Using Lemma 1 we obtain their unique most-general symbolic unifier $\sigma^{\pi_1}_{\pi}$, whose codomain is $T_\Sigma(var(\pi_1) \uplus var(\pi))$. Let then $\eta : var(\pi_1) \uplus var(\pi) \to \mathcal{T}$ be the valuation such that $\sigma^{\pi_1}_{\pi}\eta = \rho|_{var(\pi_1)} \uplus \mu|_{var(\pi)}$. We extend $\sigma^{\pi_1}_{\pi}$ to $var(\varphi, \alpha)$ by letting it be the identity on $var(\varphi, \alpha) \setminus var(\pi_1, \pi)$, and extend η to $var(\varphi, \alpha)$ such that $\eta|_{var(\phi_1,\phi_2,\pi_2)\setminus var(\pi_1)} = \rho|_{var(\phi_1,\phi_2,\pi_2)\setminus var(\pi_1)}$ and $\eta|_{var(\phi)\setminus var(\pi)} = \mu|_{var(\phi)\setminus var(\pi)}$. With these extensions we have $x(\sigma^{\pi_1}_{\pi}\eta) = x(\rho \uplus \mu)$ for all $x \in var(\varphi, \alpha)$.

Let $\varphi' \triangleq \pi_2\sigma^{\pi_1}_{\pi} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma^{\pi_1}_{\pi}$: we have the transition $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ by Definition 4. There remains to prove $\gamma' \in [\![\varphi']\!]$.

- on the one hand, $(\pi_2\sigma^{\pi_1}_{\pi})\eta = \pi_2(\sigma^{\pi_1}_{\pi}\eta) = \pi_2(\rho \uplus \mu) = \pi_2\rho = \gamma'$; thus, $(\gamma', \eta) \models \pi_2\sigma^{\pi_1}_{\pi}$;
- on the other hand,

$$\begin{array}{rl} \eta \models ((\phi \wedge \phi_1 \wedge \phi_2)\sigma^{\pi_1}_{\pi}) & \text{iff} \\ (\sigma^{\pi_1}_{\pi}\eta) \models (\phi \wedge \phi_1 \wedge \phi_2) & \text{iff} \\ (\rho \uplus \mu) \models (\phi \wedge \phi_1 \wedge \phi_2) & \text{iff} \\ \mu \models \phi \text{ and } \rho \models \phi_1 \text{ and } \rho \models \phi_2 & \end{array}$$

  Since the last relations hold by the hypotheses, it follows $\eta \models (\phi \wedge \phi_1 \wedge \phi_2)\sigma^{l}_{\pi}$. The following property was used above: if $\rho : Var \to \mathcal{T}$ is a valuation and $\sigma : Var \to T_\Sigma(Var)$ a substitution, then $\rho \models \varphi\sigma$ iff $\sigma\rho \models \varphi$.

The two above items imply $(\gamma', \eta) \models \pi_2\sigma^{\pi_1}_{\pi} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma^{\pi_1}_{\pi}$, i.e., $(\gamma', \eta) \models \varphi'$, which concludes the proof. □

Corollary 1 *For every concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ there is a symbolic execution $[\varphi_0]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_1]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_n]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ such that $\gamma_i \in [\![\varphi_i]\!]$ for $i = 0, 1, \ldots$.*

3.4 Precision

Lemma 3 *If $\gamma' \in [\![\varphi']\!]$ and $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ then there exists $\gamma \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\varphi]\!]$.*

Proof From $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ with $\varphi \triangleq \pi \wedge \phi$ and $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ we obtain $\varphi' = \pi_2\sigma^{\pi_1}_{\pi} \wedge \phi'\sigma^{\pi_1}_{\pi}$ for some ϕ'_1 such that $\models \phi'\sigma^{\pi_1}_{\pi} \leftrightarrow (\phi \wedge \phi_1 \wedge \phi_2)\sigma^{\pi_1}_{\pi}$.

From $\gamma' \in [\![\varphi']\!]$ we obtain $\eta : Var \to \mathcal{T}$ such that $\gamma' = (\pi_2\sigma^{\pi_1}_{\pi})\eta$ and $\eta \models (\phi'\sigma^{\pi_1}_{\pi})$, the latter of which, by the above equivalence, means $\eta \models ((\phi \wedge \phi_1 \wedge \phi_2)\sigma^{\pi_1}_{\pi})$. We extend $\sigma^{\pi_1}_{\pi}$ to $var(\varphi, \alpha)$ by letting it be the identity on $var(\varphi, \alpha) \setminus var(\pi_1, \pi)$. Let $\rho : Var \to \mathcal{T}$ be defined by $x\rho = x(\sigma^{\pi_1}_{\pi}\eta)$ for all $x \in var(\varphi, \pi_1)$, and $x\rho = x\eta$ for all $x \in Var \setminus var(\varphi, \pi_1)$, and let $\gamma \triangleq \pi_1\rho$.

From $\gamma' = (\pi_2\sigma_\pi^{\pi_1})\eta$ and the definition of ρ we obtain $\gamma' = \pi_2\rho$. From $\eta \models ((\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1})$ we get $\sigma_\pi^l\eta \models \phi_1$ and $\sigma_\pi^l\eta \models \phi_2$, i.e., $\rho \models \phi_1$ and $\rho \models \phi_2$, which together with $\gamma \triangleq \pi_1\rho$ and $\gamma' = \pi_2\rho$ gives $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. There remains to prove $\gamma \in [\![\varphi]\!]$.

- From $\gamma = \pi_1\rho$ using the definition of ρ we get $\gamma = \pi_1\rho = \pi_1(\sigma_\pi^{\pi_1}\eta) = (\pi_1\sigma_\pi^{\pi_1})\eta = (\pi\sigma_\pi^{\pi_1})\eta = \pi(\sigma_\pi^{\pi_1}\eta) = \pi\rho$;
- From $\eta \models ((\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1})$ and $(\eta \models (\phi\sigma_\pi^{\pi_1})$ iff $\sigma_\pi^{\pi_1}\eta \models \phi)$ we get $\rho \models \phi$.

Since $\varphi \triangleq \pi \wedge \phi$, the last two items imply $(\gamma, \rho) \models \varphi$, i.e., $\gamma \in [\![\varphi]\!]$, which completes the proof. □

We call a symbolic execution *feasible* if all its patterns are satisfiable (a pattern φ is satisfiable if there is a configuration γ such that $\gamma \in [\![\varphi]\!]$).

Corollary 2 *For every feasible symbolic execution* $[\varphi_0]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi_1]_\sim \cdots \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi_n]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} \cdots$ *there is a concrete execution* $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ *such that* $\gamma_i \in [\![\varphi_i]\!]$ *for* $i = 0, 1, \ldots$.

4 Implementation

We now describe our implementation of symbolic execution. We first show how to encode symbolic execution in a language-definition framework where rules are applied by standard rewriting, instead of the symbolic rewriting that we used up to now. Then we describe our prototype implementation in $\mathbb{K}$ of symbolic execution based on this encoding, and illustrate it on examples.

4.1 Symbolic Execution by Standard Rewriting

In this section we show how to achieve symbolic execution in a language-definition framework where rules are applied by standard rewriting. This forms the basis of our prototype implementation of symbolic execution in the $\mathbb{K}$ framework, which will be presented in the next section.

Assumption 1 *We hereafter assume that for all elementary patterns* φ*, all the variables the set* φ *have data sorts. That is, symbolic execution deals with symbolic variables only but not with, e.g., symbolic code. Using this assumption, we can assume that for each rule* $\varphi \Rightarrow \varphi' \in \mathcal{S}$*, with* $\varphi \triangleq \pi \wedge \phi$*, the term* π *is linear and all its data subterms are variables. The generality is not restricted as patterns can always be transformed to satisfy this constraint (cf. Remark 1)*

The symbolic semantics of programming languages is given by Definition 4. But this definition requires rules to be applied in a symbolic manner. The question that arises is then: how to emulate this symbolic rewriting in a setting where only standard rewriting is available, such as our generic language definition framework where languages are defined by triples $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$, of which the $\mathbb{K}$ framework is an instance? The answer is to *transform* a language definition $\mathcal{L}$ into another language definition $\mathcal{L}^{\mathfrak{s}}$, such that standard rewriting in $\mathcal{L}^{\mathfrak{s}}$ coincides with symbolic rewriting in $\mathcal{L}$. Then, one can use $\mathcal{L}^{\mathfrak{s}}$ to perform symbolic execution using standard rewriting.

We now define the components of $\mathcal{L}^{\mathfrak{s}}$. The signature $\Sigma^{\mathfrak{s}}$ is Σ extended with a new sort *Cond* with constructors for Φ-formulas and a new sort $Cfg^{\mathfrak{s}}$ (for the the symbolic configurations) with the constructor $\_\wedge^{\mathfrak{s}}\_ : Cfg \times Cond \to Cfg^{\mathfrak{s}}$. This leaves $\Sigma^{\mathfrak{s},\mathsf{Data}}$ unchanged: $\Sigma^{\mathfrak{s},\mathsf{Data}} = \Sigma^{,\mathsf{Data}}$.

The data domain $\mathcal{D}^{\mathfrak{s}}$ is the set of terms $T_{\Sigma,\mathsf{Data}}(Var)$. The model $\mathcal{T}^{\mathfrak{s}}$ is then the set of ground terms over $(\Sigma^{\mathfrak{s}} \setminus \Sigma^{\mathfrak{s},\mathsf{Data}}) \cup \mathcal{D}^{\mathfrak{s}}$ as required by our language-definition framework in Section 2.2.

The set of rules $\mathcal{S}^{\mathfrak{s}}$ includes a rule of the form $\pi_1 \wedge^{\mathfrak{s}} \psi \Rightarrow \pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2)$ for each rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ in $\mathcal{S}$, where ψ is fresh a variable of sort *Cond*.

We still have to define the set of predicates $\Pi^{\mathfrak{s}}$. Note that the predicates in Π were transformed into terms of sort *Cond*, hence, those ones do not belong to $\Pi^{\mathfrak{s}}$. Here we have the freedom to choose $\Phi^{\mathfrak{s}}$ such that $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ coincides either with $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$, or with the subset of $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ corresponding to the feasible executions, or with a subset of $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ that approximates that of the feasible executions.

For the beginning we consider $\Pi^{\mathfrak{s}} = \emptyset$. For an elementary pattern $\varphi \triangleq \pi \wedge \phi$, we let $\varphi^{\mathfrak{s}}$ denote the symbolic configuration $\varphi^{\mathfrak{s}} \triangleq \pi \wedge^{\mathfrak{s}} \phi$, and reciprocally, for each symbolic configuration $\varphi^{\mathfrak{s}} \triangleq \pi \wedge^{\mathfrak{s}} \phi$ we let $\varphi \triangleq \pi \wedge \phi$, we let $\varphi^{\mathfrak{s}}$ be the corresponding elementary pattern.

Proposition 1 $\varphi^{\mathfrak{s}} \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \varphi'^{\mathfrak{s}}$ *iff* $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi]'_\sim$

Proof We recall that, by Assumption 1, all the free variables occuring in φ', φ' (and thus also in $\varphi^{\mathfrak{s}}, \varphi'^{\mathfrak{s}}$) are of Data sorts. This assumption will be used in both directions of the proof.

($\Rightarrow$) Assume $\varphi^{\mathfrak{s}} \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \varphi'^{\mathfrak{s}}$. Then, there is a rule $\pi_1 \wedge^{\mathfrak{s}} \psi \Rightarrow \pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2) \in \mathcal{S}^{\mathfrak{s}}$, generated from $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$, and a valuation $\rho^{\mathfrak{s}} : Var \cup \{\psi\} \to \mathcal{D}^{\mathfrak{s}}$ such that $(\pi_1 \wedge^{\mathfrak{s}} \psi)\rho^{\mathfrak{s}} = \varphi^{\mathfrak{s}}$ and $(\pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2))\rho^{\mathfrak{s}} = \varphi'^{\mathfrak{s}}$. Let $\varphi^{\mathfrak{s}} \triangleq \pi \wedge^{\mathfrak{s}} \phi$, $\varphi'^{\mathfrak{s}} \triangleq \pi' \wedge^{\mathfrak{s}} \phi'$ and let $\sigma : Var \to \mathcal{T}_{\Sigma,\mathsf{Data}}(Var)(= \mathcal{D}^{\mathfrak{s}})$ denote a substitution such that $x\sigma = x\rho^{\mathfrak{s}}$ for all $x \in Var$. From $(\pi_1 \wedge^{\mathfrak{s}} \psi)\rho^{\mathfrak{s}} = \varphi^{\mathfrak{s}}$ we get $\pi_1\sigma = \pi$ and $\psi\rho^{\mathfrak{s}} = \phi$. From the above and $(\pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2))\rho^{\mathfrak{s}} = \varphi'^{\mathfrak{s}}$ we get $\pi_2\sigma = \pi'$ and $\phi \wedge (\phi_1 \wedge \phi_2)\sigma = \phi'$. By Assumption 1 and Remark 2, the (unique) substituion σ matching π_1 with π can be extended to their most-general-unifier $\sigma_\pi^{\pi_1} : var(\pi_1) \uplus var(\pi) \to T_\Sigma(var(\pi_1) \uplus var(\pi))$, which moreover is the identity over $var(\varphi)$ since we can assume without restriction of generality (possibly, after renaming some variables) that $var(\varphi) \cap var(\varphi_1) = \emptyset$ Hence, we obtain $\varphi' = \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, which implies $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi]'_\sim$ by Definition 4.

($\Leftarrow$) Assume $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi]'_\sim$, with $\varphi \triangleq \pi \wedge \phi$, then, there is a rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ such that $\varphi' = \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, where $\sigma_\pi^{\pi_1}$ is the most general unifier of π and π_1. By Assumption 1and Remark 2, $\sigma_\pi^{\pi_1} : var(\pi_1) \uplus var(\pi) \to T_\Sigma(var(\pi_1) \uplus var(\pi))$ can be decomposed into a substitution $\sigma : var(\pi_1) \to \mathcal{T}_\Sigma(var(\pi))$ such that $\pi_1\sigma = \pi$, and the identity substitution over $var(\pi)$. Hence, we obtain $\varphi' = \pi_2\sigma \wedge \phi \wedge (\phi_1 \wedge \phi_2)\sigma$. Let then $\rho^{\mathfrak{s}} :: Var \cup \{\psi\} \to \mathcal{T}^{\mathfrak{s}}$ be any valuation such that $x\sigma = x\rho^{\mathfrak{s}}$ for all $x \in var(\pi_1)$ and $\psi\rho^{\mathfrak{s}} = \phi$. We obtain $(\pi_1 \wedge^{\mathfrak{s}} \psi)\rho^{\mathfrak{s}} = \varphi^{\mathfrak{s}}$ and $(\pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2))\rho^{\mathfrak{s}} = \varphi'^{\mathfrak{s}}$, which means $\varphi^{\mathfrak{s}} \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \varphi'^{\mathfrak{s}}$ and completes the proof. □

The feasible symbolic executions can be obtained as executions of another slightly different definition of $\mathcal{L}^{\mathfrak{s}}$, at least at theoretical level, by considering $\Pi^{\mathfrak{s}}_{Cond} = \{sat\}$ and $\Pi^{\mathfrak{s}}_s = \emptyset$ for all sorts s other than *Cond*, with the interpretation $\mathcal{T}^{\mathfrak{s}}_{sat} = \{\phi \mid \phi \text{ is satisfiable in } \mathcal{T}\}$, and by taking in $\mathcal{S}^{\mathfrak{s}}$ the following conditional rules, for each $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$:

$$(\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge sat(\psi \wedge \phi_1 \wedge \phi_2) \Rightarrow \pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2)$$

Recall that $\pi_1 \wedge^{\mathfrak{s}} \psi$ and $\pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2)$ are terms in $\mathcal{L}^{\mathfrak{s}}$ and the expression $(\pi_1 \wedge \psi) \wedge sat(\psi \wedge \phi_1 \wedge \phi_2)$ is a pattern in $\mathcal{L}^{\mathfrak{s}}$, hence, the above rules are well formed.

This new definition of $\mathcal{L}^{\mathfrak{s}}$ cannot be implemented in practice because the satisfiability problem for first-order logic is undecidable, there are no find algorithms computing *sat*. To deal with this we implemented a version of $\mathcal{L}^{\mathfrak{s}}$ that approximates feasible symbolic executions. This can be done by using a predicate symbol *nsat* instead of *sat*, such that its interpretation is sound, i.e., $\mathcal{T}^{\mathfrak{s}}_{nsat} \subsetneq \{\phi \mid \phi \text{ not satisfiable in } \mathcal{T}\}$, and is computable. Then, the rules in $\mathcal{S}^{\mathfrak{s}}$ have the form

$$(\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge \neg\, unsat(\psi \wedge \phi_1 \wedge \phi_2) \Rightarrow \pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2)$$

which is what we actually implemented in our prototype tool in $\mathbb{K}$ Framework presented next.

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

In this section we present a prototype tool implementing our symbolic execution approach based on the formalisation presented so far. We first briefly present our tool and its integration within the $\mathbb{K}$ framework. In Section 4.3 we illustrate the most significant features of the tool by the means of use cases involving nontrivial languages and programs. Our tool is part of $\mathbb{K}$ [20], 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 language transformations presented in Section 4.1. 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 adds 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.

4.3 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 [11]. 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.

4.3.1 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 telling it to generate the symbolic semantics of the language by applying the transformations described in Section 4.1.

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.

```
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`

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 [8]. 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 [**?**].

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 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):

$$\texttt{\$ krun log.imp -cPC="}a >_{Int} 0 \ \wedge_{Bool} \ a <_{Int} 10\texttt{" -cIN="}a\texttt{"}$$
$$\texttt{-ltlmc "}\Box_{Ltl}\,(\texttt{L} \rightarrow_{Ltl} \texttt{logInv(a, x, k))"}$$

The $\mathbb{K}$ runner executes the command by calling the Maude LTL model-checker for the LTL formula $\Box_{Ltl}\,(\texttt{L} \rightarrow_{Ltl} \texttt{logInv}(\texttt{a},\texttt{x},\texttt{k}))$ 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.

```
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);
}
```

Figure 6: SIMPLE program: `init-arrays`

4.3.2 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:

```
$ 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.

4.3.3 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

```
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

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.

4.4 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 4.1). 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 Section 4.1 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 generate a variable `$IN`, initialized at runtime by `krun` with the value of the option `-cIN`.

5 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] . 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`). The work presented here was supported in part by Contract 161/15.06.2010, SMIS-CSNR 602-12516 (DAK).

References

[1] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. *Electr. Notes Theor. Comput. Sci.*, 144(3):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 at `http://hal.inria.fr/hal-00766220/`.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In Yi [27], pages 52–68.

[4] 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.

[5] J. de Halleux and N. Tillmann. Parameterized unit testing with Pex. In *TAP*, volume 4966 of *LNCS*, pages 171–181. Springer, 2008.

[6] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

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

[8] S. Eker, J. Meseguer, and A. Sridharanarayanan. The maude ltl model checker and its implementation. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 230–234. Springer, 2003.

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

[10] 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.

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

[12] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS'03*, pages 553–568, 2003.

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

[14] 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 *LNCS*, pages 609–615. Springer, 2011.

[15] 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.

[16] 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 *LNCS*, pages 164–181. Springer, 2004.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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. Also available as technical report `http://hdl.handle.net/2142/33771`.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] K. Yi, editor. *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *LNCS*. Springer, 2005.

Appendix: Generic Symbolic Execution with the $\mathbb{K}$ Tool

The symbolic execution framework we proposed in this paper is implemented as part of the $\mathbb{K}$ framework semantics compiler. The $\mathbb{K}$ tool includes two main components: compiler (`kompile`) and runner (`krun`). The `kompile` component compiles the operational semantics of a language into a rewrite theory. The `krun` component runs programs written in the defined language by applying the rewrite rules from the compiled semantics.

Compiling the symbolic semantics (IMP)

For this scenario, we consider the IMP language (Figure 1) enriched with I/O operations, namely `read` and `print`. Supposing that the definition of IMP is stored in a file called `imp.k`, let us compile the symbolic semantics of the language:

```
$ kompile imp.k --symbolic
```

Beside the normal compilation, the `-symbolic` flag applies the transformations described in Subsection **??** to all rules from the semantics of IMP. It also appends to the configuration two variables, namely `$IN` and `$PC`, which are used at running time, for setting the input and the initial path condition.

Running the symbolic semantics

Since the proposed transformations are meant to generalize the semantics, the normal execution of programs should not be affected. Therefore, we can run programs with concrete values using the symbolic semantics. Let us run the program from Figure 4 for `n = 10`:

```
$ krun sum.imp -cIN="ListItem(10)" -cPC="true"
Sum = 55
<path-condition>
    true
</path-condition>
<T>
    <k>
        .K
    </k>
    <in>
        #buffer ( .K )
        #istream ( 0 )
    </in>
    <out>
        #ostream ( 1 )
        #buffer ( .K )
    </out>
    <state>
        n |-> 0
        s |-> 55
    </state>
</T>
```

The `krun` tool applies the rewrite rules over the program until none of them can be applied. The command line arguments of `krun` are the filename containing the program, the input given as a list of items, and the initial path condition. The tool displays both the expected sum and

the final configuration, where the computation cell is empty, i.e. the program has been consumed completely.

Now, we run the same program with a symbolic value for `n`. A symbolic value in $\mathbb{K}$ is represented as `#sym`*Sort*`(Int)` or `#sym`*Sort*`(Id)`, where *Sort* is the sort of the symbolic value. For instance, `#symInt(0)`, `#symInt(a)`, and `#symBool(b)` are symbolic values. The result of symbolic execution is a set of a symbolic expressions, each of them corresponding to an execution path. By default, `krun` explores only one execution path:

```
$ krun sum.imp -cIN="ListItem(#symInt(n))" -cPC="true"
Sum = #symInt(n)
<path-condition>
    ((#symInt(n) >Int 0) ==Bool true) andBool
    (((#symInt(n) -Int 1) >Int 0) ==Bool false)
</path-condition>
<T>
    <k>
        .K
    </k>
    <in>
        #buffer ( .K )
        #istream ( 0 )
    </in>
    <out>
        #ostream ( 1 )
        #buffer ( .K )
    </out>
    <state>
        n |-> #symInt(n) -Int 1
        s |-> #symInt(n)
    </state>
</T>
```

As we can observe in the `<path-condition>` cell, this execution path corresponds to the case when `#symInt(n)` is 1. To get all the execution paths, we have to call `krun` with the `-search` option. Since our program contains a loop which depends on `#symInt(n)` the search space is infinite. We can search a finite sub-space by supplying the `-bound` option:

```
$ krun sum.imp -cPC="true" --search --bound 3
-cIN="ListItem(#symInt(n))"

Search results:
...

Solution 3, State 6:
<path-condition>
    (((#symInt(n) >Int 0) ==Bool true) andBool (((#symInt(n) -Int 1) >Int 0)
      ==Bool true)) andBool ((((#symInt(n) -Int 1) -Int 1) >Int 0) ==Bool
      false)
</path-condition>
<T>
    <k>
        .K
    </k>
    <in>
```

```
        #buffer ( "\n" )
    </in>
    <out>
        #buffer ( "Sum = #symInt(n) +Int #symInt(n) -Int 1\n" )
    </out>
    <state>
        n |-> (#symInt(n) -Int 1) -Int 1
        s |-> #symInt(n) +Int (#symInt(n) -Int 1)
    </state>
</T>
```

The bound value `3` instructs the tool to stop after it finds 3 complete executions. We shown only one solution here, which corresponds to the case when `#symInt(n)` is 2. The computed sum in this case is 3.

Verifying LTL formulas

Since Maude's LTL model-checker is plugged in the $\mathbb{K}$ tool, we can verify LTL formulas over programs. The $\mathbb{K}$ semantics of a language can be modified following the methodology described in [**?**], such that it supports verification of LTL properties. The methodology implies the inclusion of atomic LTL propositions, and their satisfaction by configurations.

We have modified the IMP semantics accordingly and we proved an invariant for the program computing the binary logarithm of a given number, shown in Figure 5. First, we run the program using `-search`:

```
$ krun log.imp -cPC="#symInt(a) >Int 0 andBool #symInt(a) <Int 10"
-cIN="ListItem(#symInt(a))" --search
Search results:

Solution 1, State 2:
...
Solution 2, State 6:
...
Solution 3, State 10:
...
Solution 4, State 13:
<path-condition>
 (((((#symInt(a) >Int 0) andBool (#symInt(a) <Int 10)) andBool ((#symInt(a
  ) >Int 1) ==Bool true)) andBool (((#symInt(a) divInt 2) >Int 1) ==Bool
  true)) andBool ((((#symInt(a) divInt 2) divInt 2) >Int 1) ==Bool true))
  andBool (((((#symInt(a) divInt 2) divInt 2) divInt 2) >Int 1) ==Bool false)
</path-condition>
<T>
    <k>
        .K
    </k>
    <in>
        #buffer ( "\n" )
    </in>
    <out>
        #buffer ( .K )
    </out>
    <state>
        a |-> #symInt(a)
```

```
      k |-> 3
      x |-> ((#symInt(a) divInt 2) divInt 2) divInt 2
   </state>
</T>
```

Note that the initial path conditions sets `#symInt(a)` to be greater than 0, because the logarithm is not defined for numbers ≤ 0, and less than 10 to bound the search space. The solutions above correspond to all possible values of logarithm function applied to first nine numbers, hence `k` can be 0, 1, 2, or 3. The invariant we want to prove can be deduced observing the relation between `k` and `x`. By running the program with the `-search` option, we observed that `x` is always mapped to $\texttt{a}/2^{\texttt{k}}$ and, more generally, that $\texttt{x} * 2^{\texttt{k}} \leq \texttt{a} < (\texttt{x}+1) * 2^{\texttt{k}}$. We check this invariant property by encoding it in $\mathbb{K}$ as an LTL atomic proposition `logInv(a, x, k)` and then execute the `krun` command as follows:

```
$ krun log.imp -cPC="#symInt(a) >Int 0 andBool #symInt(a) <Int 10"
-ltlmc "[]Ltl (L ->Ltl logInv(a, x, k))" -cIN="ListItem(#symInt(a))"
true
```

The LTL model-checker is called using the `-ltlmc` option. The `[]Ltl` stands for the *always* LTL operator $\Box$, and `->Ltl` for LTL implication operator. The statement label `L` denotes the LTL atomic proposition that is satisfied by the current configuration whenever the statement to be executed is labelled by `L`. Since the tool returns true, the property holds for the bounded state space.

Symbolic Execution of programs with Arrays (SIMPLE)

In the context of symbolic execution, arrays can store symbolic components, have symbolic lengths, or be themselves symbolic. For the $\mathbb{K}$ builtin arrays we have implemented the basic theory of arrays characterized by the *select-store* axioms proposed by J. McCarthy in [**?**]. In this scenario we use the SIMPLE language and the program shown in Figure 6.

Running the program giving as input an array with symbolic length and a sequence of three symbolic values, we obtain seven solutions:

```
$ krun init-arrays.simple -cPC="#symInt(n) >Int 0" --search
-cIN="ListItem(#symInt(n)) ListItem(#symInt(j)) ListItem(#symInt(x))
ListItem(#symInt(a1)) ListItem(#symInt(a2)) ListItem(#symInt(a3))"
Search results:

Solution 1, State 14:
...
Solution 7, State 61:
...
```

The solutions depend on the values of `#symInt(n)` and `#symInt(j)`. The input symbolic value `#symInt(n)` can take any value greater than 0, as stated by the initial path condition. The analysis reveals four cases of interest, namely when `#symInt(n)` is 1,2,3, or bigger than 3. Then, `#symInt(j)` can hold any value strictly less than `#symInt(n)`, because it is constrained so by the rules in the SIMPLE semantics. This amounts to a total of $7 (=1+2+3+1)$ solutions. Searching for the pattern `<T> <out> #buffer("error") </out> B:Bag </T>` will return no solutions, since none of the returned configurations contains `"error"` in the output cell:

```
$ krun init-arrays.simple -cPC="#symInt(n) >Int 0" --search
-cIN="ListItem(#symInt(n)) ListItem(#symInt(j)) ListItem(#symInt(x))
ListItem(#symInt(a1)) ListItem(#symInt(a2)) ListItem(#symInt(a3))"
```

```
--pattern="<T> <out> #buffer(\"error\") </out> B:Bag </T>"
Search results:
No search results
```

Symbolic Execution of Object-Oriented Programs (KOOL)

The program from Figure 8 shows an implementation of lists in the KOOL object-oriented language. The `List` class implements lists using arrays and has the common operations on lists, namely `insert`, `delete`, `copy`, and `getAt`. It also contains a method for testing the equality with another lists `eqTo`, by comparing their size and their elements. We also have a class `OrderedList` which inherits `List` and holds the sorted list of elements. This class overrides the `insert` method, because it must preserve the elements order. Note that by default, in `List`, `insert` appends an element at the end of the list and `delete` deletes the first element equal to the given value from the list.

Let us consider a `Main` method that initializes a list `l1`, copies `l1` into `l2`, inserts an input value `x` into `l1` and then it removes it. Finally, it tests the equality between `l1` and `l2`. In case the lists are not equal, the `"error"` message is printed at the standard output.

We are going to investigate two cases: (1) when both lists are instances of `List` and (2) when they are instances of `OrderedList`. We start with the first case:

```
$ krun lists.kool -cPC="true" --search \
> -cIN="ListItem(#symInt(e1)) ListItem(#symInt(e2)) ListItem(#symInt(x))" \
> --pattern "<T> <out> #buffer(\"error\") </out> B:Bag </T> PC:Bag"
Solution 1, State 50:
PC:Bag -->
<path-condition>
    ((((notBool (#symInt(e1) ==Int #symInt(x))) ==Bool false) andBool ((
      #symInt(e1) ==Int #symInt(x)) ==Bool true)) andBool ((#symInt(e1) ==Int
       #symInt(e2)) ==Bool false)) andBool ((#symInt(e1) ==Int #symInt(e2))
      ==Bool false)
</path-condition>
...
```

The command above searches through all the configurations and, eventually, returns those which contain `"error"` in the output cell. The tool returned one solution, corresponding to the case when `#symInt(e1)=#symInt(x)`, as we can observe from the path condition. We briefly describe how this error is obtained: The initial value of `l1` read from the input is (`#symInt(e1)`, `#symInt(e2)`). After inserting `#symInt(x)` it becomes (`#symInt(e1)`, `#symInt(e2)`, `#symInt(x)`). The `delete` method compares `#symInt(x)` with the values of `l1`. For the case `#symInt(x)=#symInt(e1)`, the first element is deleted and the list `l1` becomes (`#symInt(e2)`, `#symInt(x)`), which is different from `l2`. Now we consider the second case when `l1` and `l2` are instances of `OrderedList`:

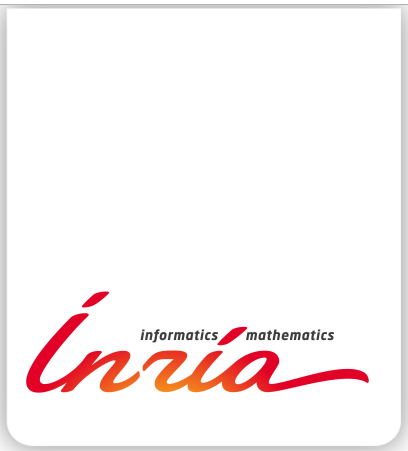
```
$ krun lists.kool -cPC="true" --search \
> -cIN="ListItem(#symInt(e1)) ListItem(#symInt(e2)) ListItem(#symInt(x))"\
> --pattern "<T> <out> #buffer(\"error\") </out> B:Bag </T> PC:Bag"
Search results:
No search results
```

In this case, the tool returns no solution, 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.

```
class List {
  int a[10];
  int size, capacity;
  void List () {
    size = 0;
    capacity = 10;
  }

  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;
    }
  }
  int getAt(int i) {
    if (i < size) {
      return a[i];
    }
  }
  bool eqTo(List l) {
    if (size != l.size) {
      return false;
    }
    int i = 0;
    while (i < size - 1 &&
           a[i] == l.getAt(i)){
      i = i+1;
    }
    if (a[i] == l.getAt(i))
      { return true; }
    else
      { return false; }
  }
  List copy() {
    List t = new List();
    int i = 0;
    t.size = size;
    while(i < size) {
      t.a[i] = a[i];
      i = i + 1;
    }
    return t;
  }
}
```

```
class OrderedList extends List {
  void OrderedList() {
    super.List();
  }

  void insert(int x) {
    if (size < capacity) {
       int i = 0;
       while(i < size &&
             a[i] <= x) {
         i = i + 1;
       }
       ++size;
       int k = size - 1;
       while(k > i) {
         a[k] = a[k-1];
         k = k - 1;
       }
       a[i] = x;
    }
  }

  OrderedList copy() {
    OrderedList t = new OrderedList();
    int i = 0;
    t.size = size;
    while(i < size) {
      t.a[i] = a[i];
      i = i + 1;
    }
    return t;
  }
}

class Main {
  void Main() {
    OrderedList l1 = new OrderedList();
    int i = 0;
    while(i < 2) {
      l1.insert(read());
      i = i + 1;
    }
    int x = read();

    List temp = l1.copy();
    OrderedList l2 = (OrderedList) temp;
    l1.insert(x);
    l1.delete(x);
    if (l2.eqTo(l1) == false) {
      print("error\n");
    }
  }
}
```

Figure 8: `lists.kool`: implementation of lists in KOOL



RESEARCH CENTRE
LILLE – NORD EUROPE

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

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

ISSN 0249-6399