A Generic Framework for Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

A Generic Framework for Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

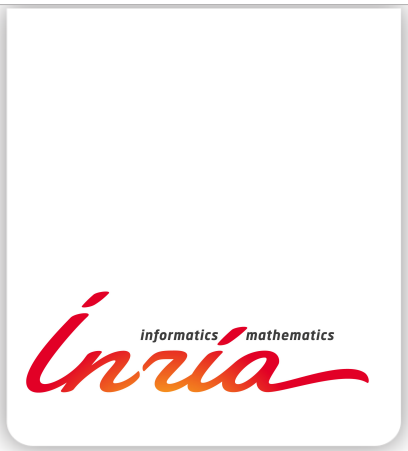
RESEARCH REPORT

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 — 32 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, meaning that the feasible symbolic executions of a program and the concrete executions of the same program mutually simulate each other. We then show how symbolic rewriting can be implemented in language definition frameworks based on standard rewriting such as the $\mathbb{K}$ framework. A prototype implementation of our approach has been developed in $\mathbb{K}$. We illustrate it on the symbolic execution, model checking, and deductive verification of 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, le model checking borné, ainsi que la vérification déductive 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 [27]. 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 containing symbolic values [32]. 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.

A symbolic program execution typically memorises symbolic values of program variables and a *path condition*, which accumulates constraints on the symbolic values on the path leading the current instruction. When the next instruction to be executed is a conditional statement, whose condition depends on symbolic values, the execution is separated into distinct branches. The path condition is then updated to distinguish between the 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 symbolic execution as the application of rewrite rules in the semantics by *symbolic rewriting*, which is a nonstandard way of applying rewrite rules. 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. Or, stated in terms of simulations: the feasible symbolic executions and the concrete executions of any given program mutually simulate each other.

On the practical side, we present a prototype implementation of our approach in $\mathbb{K}$ [37], a framework dedicated to defining formal operational semantics of languages. Since the current version of $\mathbb{K}$ is based on standard rewriting, we first show how symbolic rewriting can be achieved by applying certain modified rewrite rules (obtained by automatically transforming the original ones) in the standard manner over a symbolic domain. This relates the present formalisation of symbolic execution with an earlier approach we developed in [6]. We briefly describe our implementation and demonstrate it on nontrivial programs. The examples illustrate the bounded model checking of programs and their deductive verification with respect to Reachability-Logic (RL) [36] formulas. Deductive verification of RL formulas is based on an extension of our approach presented in detail elsewhere [7], which we only sketch here.

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.

Java PathFinder [33] 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 data structures, arrays, preconditions, and multithreading. Java PathFinder can access several Satisfiability Modulo Theories (SMT) solvers and the user can also choose

[1] Most existing operational semantics styles (small-step, big-step, reduction with evaluation contexts, ...) have been shown to be representable in this way in [41].

between multiple decision procedures. By instantiating our generic approach to the $\mathbb{K}$ definition of Java [10] we obtain some of Java PathFinder's features for free.

Another approach consists in combining concrete and symbolic execution into *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 has been implemented by several tools: DART [23], CUTE [40], EXE [12], PEX [18]. We note that our approach allows mixed concrete/symbolic execution; it can be the basis for language-independent implementations of concolic execution.

There are several tools that check program correctness using symbolic execution. Some of them are more oriented towards finding bugs [11], while others are more oriented towards verification [15, 26, 34]. Several techniques are implemented to improve the performance of these tools, such as *bounded verification* [13] and *pruning* the execution tree by eliminating redundant paths [17]. Other approaches offer support for verification of code contracts over programs. Spec# [8] is a tool developed at Microsoft that extends C# with constructs for non-null types, preconditions, postconditions, and object invariants. Spec# comes with a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks and multi-threading. A similar approach, which provides functionality for checking the correctness of a JAVA implementation with respect to a given UML/OCL specification, is the KeY [3] tool. In particular, KeY allows to prove that after running a method, its postcondition and the class invariant holds, using Dynamic Logic [24] and symbolic execution. The VeriFast tool [25] supports verification of single and multi-threaded C and Java programs annotated with preconditions and postconditions written in Separation Logic [35]. The Smallfoot tool [9, 42] uses symbolic execution together with separation logic to prove Hoare triples. There are also approaches that attempt to automatically detect invariants in programs [31, 39]. The major advantage of most of these tools is that they perform very well, being able to verify substantial pieces of code, some of which are parts of actual safety-critical systems. On the other hand, they deal only with specific programs (e.g. written using subsets of C) and specific properties (e.g., allocated memory is eventually freed).

An approach closely related to ours is implemented in the MatchC tool [36], which has beed used for verifying challenging C programs such as the Schorr-Waite garbage collector. MatchC also uses the formalism of Reachability Logic for program specifications; it is, however, dedicated to a specific language.

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 and are focused on specific applications of symbolic execution. We focus here on language-independence: given a programming language defined in an algebraic/rewriting setting, we build its symbolic semantics and use it for various analyses and verifications of programs in those languages.

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, was extended for solving reachability problems in term-rewriting systems and was applied to the analysis of security protocols [30]. Such analyses rely on powerful unification-modulo-theories algorithms [20], 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, which have arbitrarily complex datatypes.

Paper organisation After this introduction, Section 2 presents the CinK language, a kernel of the C++ programming language. We shall use CinK programs for illustrating various aspects of symbolic execution. In Section 3 we present some background theoretical material used in the rest of the paper: Reachability Logic (Section 3.1), which is used for defining operational semantics of

Exp	::=	*Id* | *Int*	
	|	`++` *Exp*	[*strict*, *prefinc*]
	|	`--` *Exp*	[*strict*, *prefdec*]
	|	*Exp* `/` *Exp*	[*strict*(*all*(*context*(*rvalue*))), *divide*]
	|	*Exp* `+` *Exp*	[*strict*(*all*(*context*(*rvalue*))), *plus*]
	|	*Exp* `>` *Exp*	[*strict*(*all*(*context*(*rvalue*)))]
Stmt	::=	*Exps* `;`	[*strict*]
	|	{*Stmts*}	
	|	`while` (*Exp*)*Stmt*	
	|	`return` *Exp* `;`	[*strict*(*all*(*context*(*rvalue*)))]
	|	`if` (*Exp*)*Stmt* `else` *Stmt*	[*strict*(*1*(*context*(*rvalue*)))]

Figure 1: Fragment of CinK syntax

languages and for stating program properties; language definitions (Section 3.2), in order to make our approach independent of the $\mathbb{K}$ language-definition framework; and unification (Section 3.3), with some technical results showing how unification can, in our setting, be implemented by matching. Section 4 contains our formalisation of symbolic execution, including the coverage and precision results stated earlier in this introduction. Section 5 briefly presents how Reachability-Logic (RL) formulas can be verified by using symbolic execution; a full treatment can be found in [7]. In Section 6 we show how symbolic execution, which uses a special kind of rewriting that we call *symbolic* rewriting, can be implemented in language definitions based on standard rewriting, such as the $\mathbb{K}$ framework; essentially, by transforming a language definition so that symbolic rewriting in the original language corresponds to standard rewriting in the transformed language. Section 7 presents a prototype tool based on the language transformations from the previous section, as well as applications of the tool for the symbolic execution, model checking, and deductive verification of nontrivial programs. We conclude in Section 8.

2 Example: the $\mathbb{K}$ Definition of the CinK language

Our running example is CinK [29], a kernel of the C++ programming language. The $\mathbb{K}$ definition of CinK can be found on the $\mathbb{K}$ Framework Github repository: `http://github.com/kframework/cink-semantics`. As any $\mathbb{K}$ definition, it consists of the language's syntax, given using a BNF-style grammar, and of its semantics, given by means of rewrite rules. In this paper we only exhibit a small part of the $\mathbb{K}$ definition of CinK, whose syntax is shown in Figure 1. Some of the grammar productions are annotated with $\mathbb{K}$-specific attributes.

A major feature of C++ expressions is the "sequenced before" relation [2], which defines a partial order over the evaluation of subexpressions. This can be easily expressed in $\mathbb{K}$ using the *strict* attribute to specify an evaluation order for an operation's operands. If the operator is annotated with the *strict* attribute then its operands will be evaluated in a nondeterministic order. For instance, all the binary operations are strict. Hence, they may induce non-determinism in programs because of possible side-effects in their arguments.

Another feature is given by the classification of expressions into *rvalues* and *lvalues*. The arguments of binary operations are evaluated as rvalues and their results are also rvalues, while, e.g., both the argument of the prefix-increment operation and its result are lvalues. The *strict* attribute for such operations has a sub-attribute *context* for wrapping any subexpression that must be evaluated as an rvalue. Other attributes (*funcall*, *divide*, *plus*, *minus*, ...) are names associated to each syntactic production, which can be used to refer to them.

$$\langle\ \langle \$PGM \rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{env}}\ \langle\cdot\rangle_{\mathsf{store}}\ \langle\cdot\rangle_{\mathsf{stack}}\ \langle\cdot\rangle_{\mathsf{return}}\ \langle\cdot\rangle_{\mathsf{out}}\ \rangle_{\mathsf{cfg}}$$

Figure 2: CinK configuration

I_1 `+` $I_2 \Rightarrow I_1 +_{Int} I_2$	[*plus*]
I_1 `/` $I_2 \wedge I_2 \neq_{Int} 0 \Rightarrow I_1 /_{Int} I_2$	[*division*]
`if(` *true* `)` St `else` $\_ \Rightarrow St$	[*if-true*]
`if(` *false* `)` $\_$ `else` $St \Rightarrow St$	[*if-false*]
`while(` B `)` $St \Rightarrow$ `if(` B `){` St `while(` B `)` St `else {}}`	[*while*]
V `;` $\Rightarrow \cdot$	[*instr-expr*]
$\langle$`++lval(` L `)` $\Rightarrow$ `lval(` L `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto (V \Rightarrow V +_{Int} 1) \cdots\rangle_{\mathsf{store}}$	[*inc*]
$\langle$`--lval(` L `)` $\Rightarrow$ `lval(` L `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto (V \Rightarrow V -_{Int} 1) \cdots\rangle_{\mathsf{store}}$	[*dec*]
$\langle\langle$`lval(` L `)`$= V \Rightarrow V \cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto \_ \Rightarrow V \cdots\rangle_{\mathsf{store}} \cdots\rangle_{\mathsf{cfg}}$	[*update*]
$\langle\langle$`$lookup(` L `)` $\Rightarrow V \cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto V \cdots\rangle_{\mathsf{store}} \cdots\rangle_{\mathsf{cfg}}$	[*lookup*]
`{` Sts `}` $\Rightarrow Sts$	[*block*]

Figure 3: Subset of rules from the K semantics of CinK

The $\mathbb{K}$ framework uses *configurations* to store program states. A configuration is a nested structure of cells, which typically include the program to be executed, input and output streams, values for program variables, and other additional information. The configuration of CinK (Figure 2) includes the $\langle\rangle_{\mathsf{k}}$ cell containing the code that remains to be executed, which 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 expression evaluations. The memory is modelled using two cells $\langle\rangle_{\mathsf{env}}$ (which holds a map from variables to addresses) and $\langle\rangle_{\mathsf{state}}$ (which holds a map from addresses to values). The configuration also includes a cell for the function call stack $\langle\rangle_{\mathsf{stack}}$ and another one $\langle\rangle_{\mathsf{return}}$ for the return values of functions. The $\langle\rangle_{\mathsf{out}}$ cell holds to output of the program and is here connected to the standard output stream.
When the configuration is initialised at runtime, a CinK program is loaded in the $\langle\rangle_{\mathsf{k}}$ cell, and all the other cells remain empty. A $\mathbb{K}$ *rule* is a topmost rewrite rule specifying transitions between configurations. Since usually only a small part of the configuration is changed by a rule, a *configuration abstraction* mechanism is used, allowing one to only specify the parts transformed by the rule. For instance, the (abstract) rule for addition, shown in Figure 3, represents the (concrete) rule

$$\begin{array}{l}\langle\langle I_1 \texttt{ + } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \\ \Rightarrow \\ \langle\langle I_1 +_{Int} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}\end{array}$$

where $+_{Int}$ is the mathematical operation for addition, I_1, I_2, V are integers, E and S are maps, and T and O are lists.

The rule for division has a side condition which restricts its application. The conditional statement `if` has two corresponding rules, one for each possible evaluation of the condition expression. The rule for the `while` loop performs an unrolling into an `if` statement. The increment and update rules have side effects in the $\langle\rangle_{\mathsf{store}}$ cell, modifying the value stored at a specific address. Finally, the reading of a value from the memory is specified by the lookup rule, which matches a value in the $\langle\rangle_{\mathsf{store}}$ and places it in the $\langle\rangle_{\mathsf{k}}$ cell. The auxiliary construct `$lookup` is used when a program variable is evaluated as an rvalue.

In addition to these rules (written by the $\mathbb{K}$ user), the $\mathbb{K}$ framework automatically generates so-called *heating* and *cooling* rules, which are induced by *strict* attributes. We show the case of

division, which is strict in both arguments:

$$A_1 \;/\; A_2 \Rightarrow \mathit{rvalue}(A_1) \curvearrowright \Box \;/\; A_2 \qquad \mathit{rvalue}(I_1) \curvearrowright \Box \;/\; A_2 \Rightarrow I_1 \;/\; A_2$$
$$A_1 \;/\; A_2 \Rightarrow \mathit{rvalue}(A_2) \curvearrowright A_1 \;/\; \Box \qquad \mathit{rvalue}(I_2) \curvearrowright A_1 \;/\; \Box \Rightarrow A_1 \;/\; I_2$$

where $\Box$ is a special symbol, destined to receive the result of an evaluation.
Example The CinK program `gcd` from Figure 4 computes the greatest common divisor of two non-negative numbers using Euclid's algorithm. We use it as an example to illustrate symbolic execution and program verification.

```
x = a;  y = b;
while (y > 0){
  r = x % y;
  x = y;
  y = r;
}
```

Figure 4: Sample CinK Program: `gcd`

□

3 Background

In this section we present some theoretical material used in the rest of the paper: Reachability Logic (Section 3.1), which is used for defining operational semantics of languages and for stating program properties; language definitions (Section 3.2), which captures the essence of language definition frameworks such as $\mathbb{K}$; and unification (Section 3.3), containing some technical results for obtaining most-general unifiers and for achieving unification via matching).

3.1 First-Order Logic, Matching Logic and Reachability Logic

A *many-sorted signature* Σ consists of a set S of sorts and of a set of $S^* \times S$-sorted set of function symbols. 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 $T_\Sigma(\mathit{Var})$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(\mathit{Var})$ denote the set of terms of sort s with variables, and $\mathit{var}(t)$ denote the set of variables occurring in the term t. For terms $t_1, \ldots, t_n$ we let $\mathit{var}(t_1, \ldots, t_n) \triangleq \mathit{var}(t_1) \cup \cdots \mathit{var}(t_n)$. For any *substitution* $\sigma : \mathit{Var} \to T_\Sigma(\mathit{Var})$ and term $t \in T_\Sigma(\mathit{Var})$ we denote by $t\sigma$ the term obtained by applying the substitution σ to t. We use the *diagrammatical order* for the composition of substitutions, i.e., for substitutions σ and σ', the composition $\sigma\sigma'$ consists in first applying σ then σ'. Let $\mathcal{T}$ be a Σ-algebra. Any *valuation* $\rho : \mathit{Var} \to \mathcal{T}$ is extended to a (homonymous) Σ-algebra morphism $\rho : T_\Sigma(\mathit{Var}) \to \mathcal{T}$. The interpretation of a ground term t in $\mathcal{T}$ is denoted by $\mathcal{T}_t$. For simplicity, we often write $\mathit{true}, \mathit{false}, 0, 1, \ldots$ instead of $\mathcal{T}_{\mathit{true}}, \mathcal{T}_{\mathit{false}}, \mathcal{T}_0, \mathcal{T}_1, \ldots$ etc.

Definition 1 (Many-Sorted First-Order Logic (FOL)) *Given a set S of sorts, a* first-order signature $\Phi = (\Sigma, \Pi)$ *consists of a $S^* \times S$-sorted set Σ of function symbols (i.e., a many-sorted signature), and a S^*-sorted set Π of predicate symbols. A* Φ-model *consists of a Σ-algebra $\mathcal{T}$ and a subset $\mathcal{T}_p \subseteq \mathcal{T}_{s_1} \times \cdots \times \mathcal{T}_{s_n}$ for each $p \in \Pi_{s_1 \ldots s_n}$. Let Var denote a S-sorted set of variables. The set of* Φ-formulas *is defined by*

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

where p ranges over predicate symbols Π, t_i range over $\Sigma(Var)$-terms, and X over finite subsets of Var. Given a Φ-formula ϕ, a Φ-model model $\mathcal{T}$, and $\rho : Var \to \mathcal{T}$, the satisfaction relation $\rho \models \phi$ is defined as follows:

1. $\rho \models \top$;
2. $\rho \models p(t_1, \ldots, t_n)$ *iff* $(t_1\rho, \ldots, t_n\rho) \in \mathcal{T}_p$;
3. $\rho \models \neg\phi$ *iff* $\rho \not\models \phi$;
4. $\rho \models \phi_1 \wedge \phi_2$ *iff* $\rho \models \phi_1$ *and* $\rho \models \phi_2$; *and*
5. $\rho \models (\exists X)\phi$ *iff there is* ρ' *with* $x\rho' = x\rho$, *for all* $x \notin X$, *such that* $\rho' \models \phi$

A formula ϕ is valid *(in $\mathcal{T}$), denoted by $\models \phi$, if it is satisfied by all valuations.*

The other first-order formulas (including disjunction, implication, equivalence, universal quantifiers,...) are defined as syntactical sugar in the usual way.

We next recall the syntax and semantics of Matching Logic (ML) and Reachability Logic (RL) [38]. ML is a static logic of configurations, whereas RL is a dynamic logic of configurations, expressing their evolution over time. RL can be used both for specifying the operational semantics of programs (e.g., the rules in Fig. 3 denote RL formulas) and as a program-specification formalism.

Definition 2 *An ML signature is first-order signature $\Phi = (\Sigma, \Pi)$, where the many-sorted algebraic signature Σ includes a sort Cfg for configurations.*

We distinguish the following particular class of ML formulas:

Definition 3 *A* basic pattern *is a term $\pi \in T_{\Sigma, Cfg}(Var)$. An* elementary pattern *is a ML formula of the form $\pi \wedge \phi$, where π is a basic pattern and ϕ is a Φ-formula, called the* condition *of the elementary pattern.*

A basic pattern π thus defines a set of (concrete) configurations, and the condition ϕ gives additional constraints these configurations must satisfy.

Definition 4 ([38]) *A ML formula is a FOL formula that allows basic patterns as predicates. Its models are pairs (γ, ρ) consisting of configurations and valuations. The satisfaction relation $\models$ between a model (γ, ρ) and a basic pattern π is defined by $\gamma = \pi\rho$. For the remaining FOL constructions they are defined as expected, e.g., $(\gamma, \rho) \models \exists X\varphi$ iff $(\gamma, \rho') \models \varphi$ for some $\rho' : Var \to \mathcal{T}$ such that $x\rho = x\rho'$ for all $x \in Var \setminus X$. If φ is a ML formula, then $[\![\varphi]\!]$ denotes the set of concrete configurations $\{\gamma \mid$ there is ρ s.t. $(\gamma, \rho) \models \varphi\}$.*

For any set of ML formulas F we let $[\![F]\!]$ denote the set $\bigcup_{\varphi \in F} [\![\varphi]\!]$.

Examples of CinK patterns are the basic pattern

$$\langle\langle I_1 \texttt{ + } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle R\rangle_{\mathsf{return}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

and the elementary pattern

$$\langle\langle I_1 \texttt{ / } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle Rt\rangle_{\mathsf{return}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$$

An example of configuration that satisfies the second pattern is

$$\langle\langle (4 \texttt{ / } 3) \curvearrowright \texttt{skip}\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto l_1\rangle_{\mathsf{env}}\langle l_1 \mapsto 5\rangle_{\mathsf{store}}\langle \cdot\rangle_{\mathsf{stack}}\langle 0\rangle_{\mathsf{return}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}.$$

For the rest of the paper we often write ML patterns in their abstract form, using the context abstraction mechanism explained in Section 3. For instance, the second pattern is simply written as $I_1 \texttt{ / } I_2 \wedge I_2 \neq_{Int} 0$. The complete definition of the pattern is obtained by the context concretisation mechanism: the code is included in the k cell and the missing cells in the configuration are added, each of them with a variable of corresponding sort inside.

Definition 5 ([38]) *A* Reachability Logic *(RL) formula is an expression of the form* $\varphi_1 \Rightarrow \varphi_2$ *where* φ_1, φ_2 *are ML formulas.*

RL is interpreted over transition systems generated by language definitions, presented in the next section where we also define the semantics of RL.

The following example illustrates RL as a program-specification formalism.

Example Consider the program `gcd` from Figure 4. The following RL formula

$$\langle\langle \texttt{gcd} \rangle_{\texttt{k}} \langle \texttt{a} \mapsto l_1 \ \texttt{b} \mapsto l_2 \rangle_{\texttt{env}} \langle l_1 \mapsto a \ l_2 \mapsto b \rangle_{\texttt{store}} \cdots \rangle_{\text{cfg}} \wedge a \geq 0 \wedge b \geq 0 \Rightarrow$$
$$\langle\langle \cdot \rangle_{\texttt{k}} \langle (\texttt{x} \mapsto l_3) \ldots \rangle_{\texttt{env}} \langle (l_3 \mapsto g) \ldots \rangle_{\texttt{store}} \cdots \rangle_{\text{cfg}} \wedge g = gcd(a,b) \quad (1)$$

specifies that after the complete execution of the program `gcd` from a configuration where the program variables `a`, `b` are bound to non-negative values a, b a configuration where `x` is bound to $gcd(a, b)$ is reached. Here, *gcd* is a mathematical description of the greatest-common-divisor (with $gcd(0, 0) = 0$ by convention). Also by convention, all variables that occur in the right-hand side of the formula, but not in its left-hand side (e.g., l_3 and g), are implicitly quantified existentially. The quantifiers are not explicitly shown, the indices from data-domain predicates such as $\geq_{Int}$ are dropped to simplify the presentation, and any content irrelevant to the formula is replaced by ellipses. □

3.2 Language Definitions

Our approach is independent of the formal framework used for defining languages as well as from the languages being defined. We thus propose a general notion of language definition based on algebraic specification and rewriting.

A language definition is a triple $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$ where Φ is a ML signature giving syntax to the language and more generally to its overall execution infrastructure, $\mathcal{T}$ is a model of Φ interpreting the various constructions therein, and $\mathcal{S}$ is a set of RL formulas defining the operational semantics of the language. These ingredients are presented in detail in the following paragraphs.

Signature A ML signature $\Phi = (\Sigma, \Pi)$ with *Cfg* the sort for configurations. Σ may also include subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all data sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{Data}$, and that terms of sort *Cfg* have subterms denoting statements (which are programs in the syntax of $\mathcal{L}$) remaining to be executed in the configuration in question.

Model A Φ-Model $\mathcal{T}$. We assume that $\mathcal{T}$ interprets the data sorts (those included in the subsignature Σ^{Data}) according to some Σ^{Data}-algebra $\mathcal{D}$, and the non-data sorts as ground terms over the signature of the form $(\Sigma \setminus \Sigma^{Data}) \cup \mathcal{D}$, i.e., the elements of $\mathcal{D}$ are added to the signature $\Sigma \setminus \Sigma^{Data}$ as constants of their respective sorts. That is, a language is parametric in its data implementation; it just assumes there is one. This is important for the existence of certain most general unifiers, discussed below. Let $\mathcal{T}_s$ denote the elements of the algebra $\mathcal{T}$ that have the sort s; the elements of $\mathcal{T}_{Cfg}$ are called *configurations*.

Rules A set $\mathcal{S}$ of RL formulas $\varphi \Rightarrow \varphi'$, with φ, φ' elementary patterns.

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. 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' *equating the fresh variables to the subterms they replaced.*

Example The basic pattern $\langle\langle C\rangle_{\mathsf{k}}\langle X\mapsto L\rangle_{\mathsf{env}}\langle L\mapsto A+_{Int}1\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathsf{cfg}}$ is nonlinear because L occurs twice in it. It also contains the non-variable data term $A+_{Int}1$. It is thus transformed into the following elementary pattern: $\langle\langle C\rangle_{\mathsf{k}}\langle X\mapsto L\rangle_{\mathsf{env}}\langle L'\mapsto A'\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathsf{cfg}} \wedge L'=_{Int}L \wedge_{Bool} A'=_{Int}A+_{Int}1$. □

A language definition generates a transition system obtained by applying the rules in the semantics $\mathcal{S}$ to configurations. The rules are applied in a standard manner for conditional rewriting (with the particularity that RL formulas have conditions in both their left and right-hand sides). That is, a valuation needs to match to left-hand side basic of the rule to the configuration, and to satisfy (both of) the rule's conditions; this generates a new configuration, obtained by applying the valuation to the right-hand side basic pattern of the rule.

Definition 6 (Transition System Generated by Language Definition) *The transition system generated by a language definition $(\Phi,\mathcal{T},\mathcal{S})$ is $(\mathcal{T}_{Cfg},\Rightarrow_{\mathcal{S}})$, where $\gamma\Rightarrow_{\mathcal{S}}\gamma'$ iff there exist $\alpha\triangleq(\varphi\Rightarrow\varphi')\in\mathcal{S}$ and $\rho:Var\to\mathcal{T}$ such that $(\gamma,\rho)\models\varphi$ and $(\gamma',\rho)\models\varphi'$. A configuration γ is* terminating *if there is no infinite path in the transition system $(\mathcal{T}_{Cfg},\Rightarrow_{\mathcal{S}})$ starting in γ.*

Remark 2 *We assume without restriction of generality that for all rules $\varphi\Rightarrow\varphi'\in\mathcal{S}$ with $\varphi\triangleq\pi\wedge\phi$, the term π is linear and all its data subterms are variables. The generality is not restricted because φ can always be replaced by a pattern φ' (cf. Remark 1) with the desired properties and $[\![\varphi]\!]=[\![\varphi']\!]$. This transformation does not modify the transition system $(\mathcal{T}_{Cfg},\Rightarrow_{\mathcal{S}})$.*

Definition 7 (Semantics of RL) *Given a language definition $(\Phi,\mathcal{T},\mathcal{S})$, a RL formula $\varphi_1\Rightarrow\varphi_2$ is* valid*, written $\mathcal{S}\models\varphi_1\Rightarrow\varphi_2$, if for all terminating configurations γ_1 and valuations ρ satisfying $(\gamma_1,\rho)\models\varphi_1$, there exist γ_2 and a (possibly empty) sequence of transitions $\gamma_1\Rightarrow_{\mathcal{S}}\cdots\Rightarrow_{\mathcal{S}}\gamma_2$ in the transition system $(\mathcal{T}_{Cfg},\Rightarrow_{\mathcal{S}})$ such that $(\gamma_2,\rho)\models\varphi_2$.*

3.3 Unification

We use unification for formally defining symbolic execution and for proving theoretical results about it. In this section we define (what we mean by) unification and prove a technical lemma which is the basis for implementing unification by matching (and, ultimately, for implementing symbolic rewriting by standard rewriting, the only kind of currently available in the $\mathbb{K}$ framework.)

We assume a given language definition $\mathcal{L}$ as in the previous section with many-sorted signature Σ and model $\mathcal{T}$. Hereafter $\uplus$ denotes disjoint union. We denote the identity substitution by *id* and the restriction of a substitution σ to a subset X of variables by $\sigma|_X$. A similar notation is used for valuations.

Definition 8 (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 often call a symbolic unifier satisfying the above a *most general unifier*, even though the standard notion of most general unifier in algebraic specifications/rewriting is slightly different. We say that terms t_1,t_2 are symbolically (resp. concretely) unifiable if they have a symbolic (resp. concrete) unifier.

Lemma 1 (Unification by Matching) *If t_1 and t_2 are terms such that*

1. *t_1 is linear, has a non-data sort, and all its data subterms are variables,*
2. *all the elements of $var(t_2)$ have data sorts, and*
3. *t_1, t_2 are concretely unifiable,*

then there exists a substitution $\sigma : var(t_1) \mapsto T_\Sigma(var(t_2))$ such that $t_1\sigma = t_2$ and such that $\sigma_{t_2}^{t_1} \triangleq \sigma \uplus id_{var(t_2)}$ is a most-general unifier of t_1, t_2.

Proof By induction on the structure of t_1. In the base case, $t_1 \in Var$ and $\sigma = (t_1 \mapsto t_2)$, thus, $\sigma_{t_2}^{t_1} = (t_1 \mapsto t_2) \uplus id|_{var(t_2)}$. Now, $\sigma_{t_2}^{t_1}$ is obviously a symbolic unifier of t_1, t_2. To show that $\sigma_{t_2}^{t_1}$ is most general, consider any concrete unifier of t_1, t_2, say, ρ. Then, (a) $t_1\sigma_{t_2}^{t_1}\rho = t_2\rho$ because $\sigma_{t_2}^{t_1}$ maps t_1 to t_2, and (b) $t_2\rho = t_1\rho$ because ρ is a concrete unifier. Thus, $t_1\sigma_{t_2}^{t_1}\rho = t_1\rho$. Moreover, for all $x \in var(t_2)$, $x\sigma_{t_2}^{t_1}\rho = x\rho$ since $\sigma_{t_2}^{t_1}$ is the identity on $var(t_2)$. Thus, for all $y \in var(t_1) \uplus var(t_2) (= \{t_1\} \uplus var(t_2))$, $y\sigma_{t_2}^{t_1}\rho = y\rho$, which proves the fact that $\sigma_{t_2}^{t_1}$ is a most general unifier (by taking $\eta = \rho$ in Definition 8 of unifiers).

For the inductive step, let $t_1 = f(t_1^1, \ldots, t_1^n)$ with $f \in \Sigma \setminus \Sigma^{Data}$, $n \geq 0$, and $t_1^1, \ldots, t_1^n \in T_\Sigma(Var)$ for $i = 1, \ldots, n$. There are two subcases regarding t_2:

- t_2 is a variable. This is impossible, since t_2 should be of a data sort because it is a variable, and of a non-data sort because of the lemma's hypotheses.
- $t_2 = g(t_2^1, \ldots, t_2^m)$ with $g \in \Sigma$, $m \geq 0$, and $t_2^1, \ldots, t_2^m \in T_\Sigma(Var)$. Let ρ be a concrete unifier of t_1, t_2, thus, $t_1\rho = f(t_1^1, \ldots, t_1^n)\rho = \mathcal{T}_f(t_1^1\rho, \ldots, t_1^n\rho) = f(t_1^1\rho, \ldots, t_1^n\rho) =_\mathcal{T} t_2\rho = \mathcal{T}_g(t_2^1\rho, \ldots, t_2^m\rho)$, 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 as ground terms, we have $f = g$, $m = n$, and $t_1^i\rho = t_2^i\rho$ for $i = 1, \ldots, n$. The respective subterms t_1^i and t_2^i of t_1 and t_2 satisfy the hypotheses of our lemma, except maybe for the fact that t_1^i may have have a data sort. There are again two cases:
  - if for some $i \in \{1, \ldots, n\}$, t_1^i has a data sort then by the hypotheses of our lemma t_1^i is a variable, and we let $\sigma^i \triangleq (t_1^i \mapsto t_2^i)$ and $\sigma_{t_2^i}^{t_1^i} \triangleq \sigma_i \uplus id_{var(t_2^i)}$, which is a most-general unifier of t_1^i and t_2^i, which is proved like in the base case;
  - otherwise, t_1^i and t_2^i satisfy all the the hypotheses of our lemma. We can then use the induction hypothesis and obtain substitutions $\sigma^i : var(t_1^i) \to T_\Sigma(var(t_2^i))$ such that $t_1^i\sigma^i = t_2^i$ for all $i = 1, \ldots, n$, and the corresponding most-general-unifiers $\sigma_{t_2^i}^{t_1^i}$ for t_1^i and t_2^i, of the form $\sigma_{t_2^i}^{t_1^i} = \sigma^i \uplus id_{var(t_2^i)}$.

  Let $\sigma \triangleq \biguplus_{i=1}^n \sigma^i : var(t_1) \to T_\Sigma(var(t_2))$, which is a well-defined substitution thanks to the linearity of t_1. We obtain $t_1\sigma = t_2$ from $t_1^i\sigma^i = t_2^i$ for all $i = 1, \ldots, n$. Thus, $\sigma_{t_2}^{t_1} \triangleq \sigma \uplus id_{var(t_2)}$ is (obviously) a symbolic unifier of t_1, t_2; we only have to prove that it is most general. For this, we first note that the equality $\sigma_{t_2}^{t_1} = \biguplus_{i=1}^n \sigma_{t_2^i}^{t_1^i}$ also holds. Then, consider any concrete unifier ρ of t_1 and t_2, thus, $t_1^i\rho = t_2^i\rho$ for $i = 1, \ldots, n$. From the fact that all the $\sigma_{t_2^i}^{t_1^i}$ are most-general-unifiers of t_1^i and t_2^i for $i = 1, \ldots, n$, we obtain the existence of valuations η_i such that $\sigma_{t_2^i}^{t_1^i}\eta_i = \rho|_{(var(t_1^i) \uplus var(t_2^i))}$, for $i = 1, \ldots, n$. Then, $\eta \triangleq \biguplus_{i=1}^n \eta_i$, which coincides with ρ on $var(t_2)$ and is well-defined on $var(t_1)$ thanks to the linearity of t_1, and η 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 3 *The most general unifier whose existence is stated by Lemma 1 is also unique, because* $\sigma_{t_2}^{t_1}$ *is defined to be* $\sigma \uplus id_{var(t_2)}$ *and* σ*, which is a (syntactical) match of* t_1 *on* t_2*, is unique when it exists.*

4 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 are the naturally expected ones from a symbolic execution framework. They allow to perform analyses on symbolic programs, and to transfer the results of those analyses to concrete instances of the symbolic programs in question.

In the rest of the paper we consider given a language definition $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$, with $\Phi = (\Sigma, \Pi)$ and *Cfg* the sort for configurations. Moreover, we assume that, for every rule $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ with $\varphi_1 \triangleq \pi_1 \wedge \phi_1$, π_1 satisfies the hypotheses regarding t_1 of Lemma 1, that is, π_1 is linear and all its data subterms are variables. This can always be obtained by rule transformations, cf. Remark 2.

Symbolic execution essentially consists of applying the semantical rules over patterns using the most general unifiers whose existence and unicity are given by Lemma 1 and Remark 3, in a nonstandard way described below.

We first define the relation $\sim$ between patterns 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. The symbolic transition relation is a relation between such equivalence classes so that, in practice, simplifications of conditions do not generate transitions.

Definition 9 (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* $\varphi \triangleq \pi \wedge \phi$*, all the variables in* $var(\pi)$ *have data sorts, there is a rule* $\alpha \triangleq \varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ *with* $\varphi_i \triangleq \pi_i \wedge \phi_i$ *for* $i = 1, 2$*,* π_1 *and* π *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. Lemma 1), extended as the identity substitution over the variables in* $var(\phi_1, \varphi_2) \setminus var(\pi, \pi_1)$*.*

We call *symbolic rewriting* the manner in which rules are applied in Def. 9.

Assumption 1 *Hereafter we assume that for all elementary patterns* $\varphi \triangleq \pi \wedge \phi$*,* $\varphi' \triangleq \pi' \wedge \phi'$ *such that* $[\varphi]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi']_\sim$*,* π *and* π' *may only have variable of data sorts. This can be obtained by starting with an initial pattern satisfying these assumptions and by ensuring that they are preserved by the rules* $\mathcal{S}$*. Thus, our symbolic execution framework allows symbolic data but no symbolic code.*

Remark 4 *The symbolic transition relation is finitely branching: each symbolic state* $[\varphi]_\sim$ *has finitely many successors by the symbolic transition relation, since there are at most finitely many rules in* $\mathcal{S}$ *that match the basic pattern of* φ*, and the (possibly, infinitely many) elementary patterns equivalent to those generated by the rules are collapsed into finitely many equivalence classes.*

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 $[\varphi]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi']_\sim$ we

assume without restriction on generality that $var(\varphi) \cap var(\alpha) = \emptyset$, which can always be obtained by variable renaming.

We now show that the symbolic execution thus defined is related with concrete execution via the coverage and precision properties stated in the introduction.

4.1 Coverage

The coverage property states that the symbolic transition system forward-simulates the concrete one.

Lemma 2 (Coverage) *If $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\varphi]\!]$ (with all variables in $var(\varphi)$ of data sorts) 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 $\gamma = \pi\mu$ 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 9. 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,

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

  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.

The next corollary follows from the previous lemma. It can be used to draw conclusions about the absence of concrete program executions on a given path from the absence of feasible symbolic executions on the same path.

Corollary 1 *For every concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$, and pattern φ_0 such that $\gamma_0 \in [\![\varphi_0]\!]$ (with all the variables in $var(\varphi_0)$ of data sorts), there exists 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 $i = 0, 1, \ldots$.*

4.2 Precision

The precision property states that the symbolic transition system is backwards-simulated by the concrete one. Forward simulation could not hold in this case, because the patterns resulting from a symbolic transition may be unsatisfiable (a pattern φ is satisfiable if there is a configuration γ such that $\gamma \in [\![\varphi]\!]$).

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 $\varphi' = \pi_2\sigma_\pi^{\pi_1} \wedge \phi'\sigma_\pi^{\pi_1}$ for some ϕ' such that $\models \phi'\sigma_\pi^{\pi_1} \leftrightarrow (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$.

From $\gamma' \in [\![\varphi']\!]$ we obtain $\eta : Var \to \mathcal{T}$ such that $\gamma' = (\pi_2\sigma_\pi^{\pi_1})\eta$ and $\eta \models (\phi'\sigma_\pi^{\pi_1})$, the latter of which, by the above equivalence, means $\eta \models ((\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1})$. We extend $\sigma_\pi^{\pi_1}$ 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 $x\rho = x(\sigma_\pi^{\pi_1}\eta)$ 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. The next corollary, which follows from the previous lemma, can be used to draw conclusions on the existence of concrete program executions on a given path from the existence of feasible symbolic executions on the same path.

Corollary 2 *For every feasible symbolic execution $[\varphi_0]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_1]_\sim \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_n]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ 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 $i = 0, 1, \ldots$.*

The corollaries in this section say that symbolic execution can be used as a sound program-analysis technique. However, symbolic execution is, in general, not enough for performing program verification, because one can (obviously) only generate bounded-length symbolic executions, whereas program verification, especially in the presence of loops and recursive function calls, would require in general executions of an unbounded length. For example, verifying the RL formula (1) on the program in Fig. 4 require such an unbounded-length symbolic executions because of the unboundedly many iterations of the loop.

5 Application: Reachability-Logic Verification

We briefly show in this section how symbolic execution can be used for verifying RL formulas. All details regarding this matter are presented in [7]. Essentially, in order to prove that a semantics $\mathcal{S}$ satisfies a set G of RL formulas called *goals*, i.e., for proving $\mathcal{S} \models G$ (i.e., $\mathcal{S} \models g$ for all $g \in G$) one can reuse the goals G as hypotheses, provided that the set of goals to be proved is replaced by another set of goals $\Delta_{\mathcal{S}}(G)$, obtained by symbolic execution from the set G. This goal-as-hypothesis reuse is essential for proving programs with unbounded execution lengths induced by loops of recursive function calls. Thus, from $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ one deduces $\mathcal{S} \models G$; we call this

implication the *circularity principle* for RL. Here, $\Vdash$ denotes the entailment of a certain proof system, presented below, in which symbolic execution also plays a major role.

The following definition of *derivative* is an essential operation in RL verification approach. It computes (up to the equivalence relation $\sim$) a disjunction of all the successors of a pattern by the symbolic transition relation. It uses the choice operation ε, which chooses an arbitrary element in a nonempty set.

Definition 10 *The* derivative $\Delta_{\mathcal{S}}(\varphi)$ *of an elementary pattern* φ *for a set* $\mathcal{S}$ *of rules is*

$$\Delta_{\mathcal{S}}(\varphi) \triangleq \bigvee_{[\varphi]_\sim \Rightarrow^{s}_{\mathcal{S}} [\varphi']_\sim} \varepsilon([\varphi']_\sim)$$

We say φ *is* derivable for $\mathcal{S}$ *if the derivative* $\Delta_{\mathcal{S}}(\varphi)$ *is a nonempty disjunction.*

Remark 5 *Since the symbolic transition relation is finitely branching (Remark 4), the derivative is a finite disjunction. Note that according to Definition 10, the derivative of a pattern* φ *can be false (i.e., the empty disjunction).*

Definition 11 (Derivable Pattern) *An elementary pattern* $\varphi \triangleq \pi \wedge \phi$ *is* derivable *for* $\mathcal{S}$ *if* $\Delta_{\mathcal{S}}(\varphi)$ *is a nonempty disjunction.*

The notion of *total* semantics is essential for the soundness of our approach. *Weakly-well defined* semantical rules are essential for soundness as well:

Definition 12 (Total Semantics) *We say that a set* $\mathcal{S}$ *of semantical rules is* total *if for each basic pattern* π_1 *occurring in the left-hand side of a rule, it holds that* $\models \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} (\phi_1 \wedge \phi_2)$*, where* $\models$ *denotes validity in* $\mathcal{T}$*.*

Definition 13 ([36]) *A set of rules* $\mathcal{S}$ *is* weakly well-defined *if for each* $\varphi \wedge \in \Rightarrow \varphi' \mathcal{S}$ *and* $\rho : Var \to \mathcal{T}$ *there exists a configuration* γ *such that* $(\gamma, \rho) \models \varphi'$*.*

Remark 6 *The semantics of CinK is not total because of the rules for division and modulo. The rule for division:* $\langle\langle I_1 \,/\, I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1 /_{Int} I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}}$ *does not meet the condition of Definition 12 because the "disjunction" in that definition reduces to* $I_2 \neq 0$*, which is not valid. The semantics can be made total by adding a rule* $\langle\langle I_1 \,/\, I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}} \wedge I_2 = 0 \Rightarrow \langle\langle error \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}}$ *that leads divisions by zero into "error" configurations. Despite this transformation, the CinK semantics is still not weakly-well-defined because the right-hand side* $\langle\langle I_1 /_{Int} I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}}$ *is not defined for valuations sending* I_2 *to* 0*. The semantics can be transformed into a total and weakly-well-defined one by merging the above rules into the single following rule:* $\langle\langle I_1 \,/\, I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}} \Rightarrow (\langle\langle I_1 /_{Int} I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0) \vee (\langle\langle error \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}} \wedge I_2 = 0)$ *We assume hereafter that the CinK semantics has been transformed as above.*

The notion of cover, defined below, is essential in situations where a proof goal is circularly used as a hypothesis. Such goals can only be used in symbolic execution only when they *cover* the pattern being symbolically executed:

Definition 14 (Cover) *Consider an elementary pattern* $\varphi \triangleq \pi \wedge \phi$*. A set of rules* $\mathcal{S}'$ *satisfying* $\models \phi \to \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} (\phi_1 \wedge \phi_2)\sigma^{\pi_1}_{\pi}$ *is a* cover *of* φ*.*

Remark 7 *The existence of the most-general unifier* $\sigma^{\pi_1}_{\pi}$ *in the above definition means that the rules in* $\mathcal{S}'$ *are unifiable with the basic pattern* π *of* φ*. In particular,* $\bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} (\phi_1 \wedge \phi_2)\sigma^{\pi_1}_{\pi}$ *is a nonempty disjunction, otherwise the validity in Definition 14 would not hold (an empty disjunction is false).*

$$\text{[SymbolicStep]}\ \frac{\varphi \textit{ derivable for } \mathcal{S}}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \Delta_{\mathcal{S}}(\varphi)}$$

$$\text{[CircularHypothesis]}\ \frac{\alpha \in G \quad \alpha \textit{ covers } \varphi}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \Delta_{\{\alpha\}}(\varphi)}$$

$$\text{[Implication]}\ \frac{\models \varphi \rightarrow \varphi'}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \varphi'}$$

$$\text{[CaseAnalysis]}\ \frac{\mathcal{S} \cup G \Vdash \varphi_1 \Rightarrow \varphi \quad \mathcal{S} \cup G \Vdash \varphi_2 \Rightarrow \varphi}{\mathcal{S} \cup G \Vdash (\varphi_1 \vee \varphi_2) \Rightarrow \varphi}$$

$$\text{[Transitivity]}\ \frac{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \varphi'' \quad \mathcal{S} \cup G \Vdash \varphi'' \Rightarrow \varphi'}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \varphi'}$$

Figure 5: Proof System for $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$.

We now have almost all the ingredients for proving RL formulas by symbolic execution. Assume a language with a semantics $\mathcal{S}$, and a finite of RL formulas with elementary patterns in their left-hand sides $G = \{\varphi_i \Rightarrow \varphi_i' \mid i = 1, \ldots, n\}$.

Definition 15 *A RL formula $\varphi \Rightarrow \varphi'$ is* derivable *for $\mathcal{S}$ if φ is derivable for $\mathcal{S}$. If G is a set of RL formulas then $\Delta_{\mathcal{S}}(G)$ denotes the set $\{\Delta_{\mathcal{S}}(\varphi) \Rightarrow \varphi' \mid \varphi \Rightarrow \varphi' \in G\}$. If Φ is a set of RL formulas, then $\mathcal{S} \vdash \Phi$ denotes the conjunction $\bigwedge_{\varphi \Rightarrow \varphi' \in \Phi}(\mathcal{S} \vdash \varphi \Rightarrow \varphi')$ and $\mathcal{S} \models \Phi$ denotes $\bigwedge_{\varphi \Rightarrow \varphi' \in \Phi}(\mathcal{S} \models \varphi \Rightarrow \varphi')$.*

Theorem 1 (Circularity Principle for RL) *Consider the proof system in Figure 5. If $\mathcal{S}$ is total and weakly well-defined, and G is derivable for $\mathcal{S}$, then $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ implies $\mathcal{S} \models G$.*

This theorem is proved in [7] by showing that, under the hypothesis of a total semantics, the rules of the $\Vdash$ proof system are derived rules of a more abstract proof system of RL introduced in [36], which is proved there to be sound under the hypothesis of weakly well-defined semantics. If the semantics of a language is not total and/or weakly well-defined one can make it so by adding additional rules for covering the missing cases and by joining all rules with the same left-hand side basic pattern into one rule whose right-hand side is the disjunction of right-hand sides of the rules it replaced as illustrated above.

Example We show how the RL formula (1) is proved using the deductive system in Figure 5, which amounts to verifying the `gcd` program. For this, we consider the two following formulas, where `while` denotes the program fragment consisting of the `while` loop, and `body` denotes the body of the loop, and in which for readability, the ellipsis stands for cells not playing significant roles, and sort indices from predicates such as $\leq_{Int}$ are dropped:

$$\begin{array}{l}
\langle\langle \texttt{while} \rangle_{\mathsf{k}} \langle \texttt{a} \mapsto l_1\ \texttt{b} \mapsto l_2\ \texttt{x} \mapsto l_3\ \texttt{y} \mapsto l_4\ \cdots \rangle_{\mathsf{env}} \langle l_1 \mapsto a\, l_2 \mapsto b\, l_3 \mapsto x\, l_4 \mapsto y\ \cdots \rangle_{\mathsf{store}} \cdots \rangle_{\mathrm{cfg}} \wedge \\
\qquad gcd(a,b) = gcd(x,y) \wedge x \geq 0 \wedge y \geq 0 \Rightarrow \\
\langle\langle \cdot \rangle_{\mathsf{k}} \langle \texttt{a} \mapsto l_1\ \texttt{b} \mapsto l_2\ \texttt{x} \mapsto l_3\ \texttt{y} \mapsto l_4\ \cdots \rangle_{\mathsf{env}} \langle l_1 \mapsto a\, l_2 \mapsto b\, l_3 \mapsto x'\, l_4 \mapsto y'\ \cdots \rangle_{\mathsf{store}} \cdots \rangle_{\mathrm{cfg}} \wedge \\
\qquad gcd(a,b) = gcd(x',y') \wedge x' \geq 0 \wedge y' = 0 \qquad (2) \\
\langle\langle \texttt{body} \rangle_{\mathsf{k}} \langle \texttt{a} \mapsto l_1\ \texttt{b} \mapsto l_2\ \texttt{x} \mapsto l_3\ \texttt{y} \mapsto l_4\ \cdots \rangle_{\mathsf{env}} \langle l_1 \mapsto a\, l_2 \mapsto b\, l_3 \mapsto x\, l_4 \mapsto y\ \cdots \rangle_{\mathsf{store}} \cdots \rangle_{\mathrm{cfg}} \wedge \\
\qquad gcd(a,b) = gcd(x,y) \wedge x > 0 \wedge y \geq 0 \Rightarrow \\
\langle\langle \cdot \rangle_{\mathsf{k}} \langle \texttt{a} \mapsto l_1\ \texttt{b} \mapsto l_2\ \texttt{x} \mapsto l_3\ \texttt{y} \mapsto l_4\ \cdots \rangle_{\mathsf{env}} \langle l_1 \mapsto a\, l_2 \mapsto b\, l_3 \mapsto x'\, l_4 \mapsto y'\ \cdots \rangle_{\mathsf{store}} \cdots \rangle_{\mathrm{cfg}} \wedge \\
\qquad gcd(a,b) = gcd(x',y') \wedge x' \geq 0 \wedge y' \geq 0 \qquad (3)
\end{array}$$

The rules say that the `while` loop and its body preserve an invariant: the *gcd* of the (constant) values of `a`, `b` equals the *gcd* of the current values of `x`, `y`.

- the formula (1) is proved by applying a number of times the SymbolicStep rule until the k cell contains the `while` program fragment. Then, the CircularHypothesis rule is applied with the formula (2). Next, the Implication rule is used to prove that the pattern resulting from applying the formula (2) implies the right-hand side of the formula (1). Finally, the Transitivity rule builds a proof of (1) from the individual rule applications described above.
- the formula (2) is proved by applying a number of times the SymbolicStep and CaseAnalysis rules, until the program remaining to be executed is either:
  - the empty program: then, the Implication rule is used to prove that the current pattern implies the right-hand side of the formula (2), and then Transitivity builds a proof of (2) from these individual rule applications;
  - the `body` program (i.e the loop's body), possibly followed by some other program (as denoted by the dots in the k cell). In this case, CircularHypothesis is applied with the formula (3), then Implication is used to prove that the current pattern implies the right-hand side of the formula (2), and finally Transitivity builds a proof of (2) from these rule applications.
  - the formula (3) is proved by applying a number of times the SymbolicStep rule until the k cell contains the `while` program fragment once again. From this point on, the proof is completely similar to the proof of (1).

This concludes the proof of the set of goals (1), (2), and (3), and, in particular, of the fact that `gcd` meets its specification (1). Note how the proofs of all goals have used symbolic execution as well other goals as circular hypotheses. □

6 Symbolic Execution via Language Transformation

In this section we show how to achieve symbolic execution in language-definition frameworks, such as the $\mathbb{K}$ 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.

The symbolic semantics of programming languages is given by Definition 9. But this definition requires rules to be applied in a symbolic manner. A question that arises is then: how to implement this symbolic rewriting in a setting where only standard rewriting is available, such as our generic language definition framework, where languages are triples $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$? The answer is to *transform* a language definition $\mathcal{L}$ into another language definition $\mathcal{L}^{\mathfrak{s}}$, such that standard rewriting in $\mathcal{L}^{\mathfrak{s}}$ corresponds to symbolic rewriting in $\mathcal{L}$.

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 $T_{\Sigma,\mathsf{Data}}(Var|_{\mathsf{Data}})$, where $Var|_{\mathsf{Data}}$ denotes the subset of variables of data sort. 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 introduced in Section 3.

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, Π is not included in $\Pi^{\mathfrak{s}}$. Here we have the freedom to choose $\Pi^{\mathfrak{s}}$

such that $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$ corresponds to $\Rightarrow^\mathfrak{s}_\mathcal{S}$ (cf. Proposition 1), or to the subset of $\Rightarrow^\mathfrak{s}_\mathcal{S}$ corresponding to the feasible executions, or to a subset of $\Rightarrow^\mathfrak{s}_\mathcal{S}$ that approximates the feasible executions.

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

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*.

The first implication un Proposition 1 below says that for each transition in $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$ there is a transition in $\Rightarrow^\mathfrak{s}_\mathcal{S}$ that is "more general" than the one in $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$. and the second implication says that for each transition in $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$ there exists essentially the same transition in $\Rightarrow^\mathfrak{s}_\mathcal{S}$ (up to equivalent pattern conditions).

Proposition 1 (relating $\Rightarrow^\mathfrak{s}_\mathcal{S}$ and $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$)

1. *If $\widehat{\varphi}^\mathfrak{s} \Rightarrow_{\mathcal{S}^\mathfrak{s}} \widehat{\varphi'}^\mathfrak{s}$ then there exist a substitution σ and patterns φ and φ' such that $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$ and $\widehat{\varphi} = \varphi\sigma, \widehat{\varphi'} = \varphi'\sigma$.*
2. *If $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$ then $\widehat{\varphi}^\mathfrak{s} \Rightarrow_{\mathcal{S}^\mathfrak{s}} \widehat{\varphi'}^\mathfrak{s}$ for some $\widehat{\varphi} \in [\varphi]_\sim$, $\widehat{\varphi'} \in [\varphi']_\sim$.*

Proof We recall that, by Assumption 1, all the free variables occuring in the basic patterns of φ, φ' (and thus also in $\varphi^\mathfrak{s}, \varphi'^\mathfrak{s}$ and in all patterns $\sim$-equivalent to φ, φ') are of data sorts. This is used in the proof to apply Lemma 1.

(1) Assume $\widehat{\varphi}^\mathfrak{s} \Rightarrow_{\mathcal{S}^\mathfrak{s}} \widehat{\varphi'}^\mathfrak{s}$, where $\widehat{\varphi} \triangleq \pi \wedge \phi$. 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 $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$, and $\rho^\mathfrak{s} : \mathit{Var} \cup \{\psi\} \to \mathcal{T}^\mathfrak{s}$ such that $(\pi_1 \wedge^\mathfrak{s} \psi)\rho^\mathfrak{s} = \widehat{\varphi}^\mathfrak{s}$ and $(\pi_2 \wedge^\mathfrak{s} (\psi \wedge \phi_1 \wedge \phi_2))\rho^\mathfrak{s} = \widehat{\varphi'}^\mathfrak{s}$. From $(\pi_1 \wedge^\mathfrak{s} \psi)\rho^\mathfrak{s} = \widehat{\varphi}^\mathfrak{s}$ we obtain $\pi = \pi_1\rho^\mathfrak{s}$ and $\phi = \psi\rho^\mathfrak{s}$. Assuming without restriction of generality (possibly, after renaming some variables) that $\mathit{var}(\widehat{\varphi}^\mathfrak{s}) \cap \mathit{var}(\pi_1 \wedge \phi_1) = \emptyset$, it follows that the restriction of $\rho^\mathfrak{s}$ to $\mathit{var}(\pi_1, \pi)$ is the same with the most-general-unifier $\sigma^{\pi_1}_\pi$ given by Lemma 1. Let $\varphi \triangleq \widehat{\varphi}$ and $\varphi' \triangleq (\pi_2 \wedge \phi \wedge \phi_1 \wedge \phi_2)\sigma^{\pi_1}_\pi$. We obviously have $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$ by Definition 9. We recall that $\sigma^{\pi_1}_\pi$ is extended as identity over $\mathit{var}(\alpha, \widehat{\varphi}^\mathfrak{s}) \setminus \mathit{var}(\pi, \pi_1)$ in Definition 9. Therefore we consider the substitution σ defined by $x\sigma = x\rho^\mathfrak{s}$ for $x \in \mathit{var}(\alpha, \widehat{\varphi}^\mathfrak{s}) \setminus \mathit{var}(\pi, \pi_1)$ and as the identity in the rest. We obtain $\varphi'\sigma = \widehat{\varphi'}$ and $\varphi\sigma = \widehat{\varphi}\sigma = \widehat{\varphi}$ by the definition of σ, which concludes the first implication of the proposition.

(2) Assume $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$. Then, there are $\widehat{\varphi} \in [\varphi]_\sim$, $\widehat{\varphi'} \in [\varphi']_\sim$, with $\widehat{\varphi} \triangleq \pi \wedge \widehat{\phi}$ and $\models \widehat{\phi} \leftrightarrow \phi$, and a rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ such that $\widehat{\varphi'} = \pi_2\sigma^{\pi_1}_\pi \wedge (\widehat{\phi} \wedge \phi_1 \wedge \phi_2)\sigma^{\pi_1}_\pi$, where $\sigma^{\pi_1}_\pi$ is the most general unifier of π and π_1, cf Lemma 1. This Lemma also says that $\sigma^{\pi_1}_\pi$ can be decomposed into a substitution σ over $\mathit{var}(\pi_1)$ such that $\pi_1\sigma = \pi$, and the identity substitution elsewhere. Since $\mathit{var}(\pi) \cap \mathit{var}(\pi_1) = \emptyset$ we can, possibly by renaming variables, obtain that $\mathit{var}(\widehat{\varphi}) \cap \mathit{var}(\pi_1) = \emptyset$ and thus σ has no effect on $\widehat{\phi}$. Hence, we obtain $\widehat{\varphi'} = \pi_2\sigma \wedge \widehat{\phi} \wedge (\phi_1 \wedge \phi_2)\sigma$. Let then $\rho^\mathfrak{s} : \mathit{Var} \cup \{\psi\} \to \mathcal{T}^\mathfrak{s}$ be any valuation such that $x\sigma = x\rho^\mathfrak{s}$ for all $x \in \mathit{var}(\pi_1)$ and $\psi\rho^\mathfrak{s} = \widehat{\phi}$. We obtain $(\pi_1 \wedge^\mathfrak{s} \psi)\rho^\mathfrak{s} = \widehat{\varphi}^\mathfrak{s}$ and $(\pi_2 \wedge^\mathfrak{s} (\psi \wedge \phi_1 \wedge \phi_2))\rho^\mathfrak{s} = \widehat{\varphi'}^\mathfrak{s}$, which means that the transition $\widehat{\varphi}^\mathfrak{s} \Rightarrow_{\mathcal{S}^\mathfrak{s}} \widehat{\varphi'}^\mathfrak{s}$ is generated by the rule $\pi_1 \wedge^\mathfrak{s} \psi \wedge \in \Rightarrow \pi_2 \wedge^\mathfrak{s} (\psi \wedge \phi_1 \wedge \phi_2)\mathcal{S}^\mathfrak{s}$ and the valuation $\rho^\mathfrak{s}$, which completes the second implication and the proof of the proposition.

Remark 8 *A coverage result (cf. Section 4.1) relating $\Rightarrow_\mathcal{S}$ and $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$ immediately follows from the original coverage result and the second implication of Proposition 1. Indeed, assume $\gamma \Rightarrow_\mathcal{S} \gamma'$ and $\gamma \in [\![\widehat{\varphi}^\mathfrak{s}]\!]$, where $[\![\widehat{\varphi}^\mathfrak{s}]\!] \triangleq [\![\widehat{\varphi}]\!]$. Since φ and $\widehat{\varphi}$ differ at most by their (equivalent) conditions, it follows that $\gamma \in [\![\varphi]\!]$, which, together with $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$, implies, thanks to the original coverage result, that $\gamma' \in [\![\varphi']\!]$. Since φ' and $\widehat{\varphi'}$ differ at most by their (logically equivalent) conditions we obtain $\gamma' \in [\![\widehat{\varphi'}^\mathfrak{s}]\!]$, i.e., the coverage of $\Rightarrow_\mathcal{S}$ by $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$.*

Conversely, a precision result relating $\Rightarrow_\mathcal{S}$ by $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$ also follows from the original precision result and the first implication of Proposition 1, by using the fact (taken from its proof) that

$\varphi = \widehat{\varphi}$. Indeed, assume $\widehat{\varphi}^{\mathfrak{s}} \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \widehat{\varphi}'^{\mathfrak{s}}$ and $\gamma' \in [\![\widehat{\varphi}'^{\mathfrak{s}}]\!]$. Since $\widehat{\varphi}' = \varphi\sigma$ it follows that $\gamma' \in [\![\varphi']\!]$ (i.e., φ being more general than $\widehat{\varphi}'$, it contains all the instances of $\widehat{\varphi}'$, including the configuration γ'). Using $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ and the original precision result, we obtain a configuration $\gamma \in [\![\varphi]\!]$ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$, and since the proof of the first implication gave $\varphi = \widehat{\varphi}$, we have $\gamma \in [\![\widehat{\varphi}^{\mathfrak{s}}]\!]$, proving the precision of $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ with respect to $\Rightarrow_{\mathcal{S}}$.

The coverage and precision results relating $\Rightarrow_{\mathcal{S}}$ and $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ are important because they say that $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ has the natural properties expected from symbolic execution.

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 $\pi_1 \wedge^{\mathfrak{s}} \psi$, $\pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2)$ are terms in $\mathcal{L}^{\mathfrak{s}}$, thus, the expression $(\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge sat(\psi \wedge \phi_1 \wedge \phi_2)$ is a pattern and the above rule is 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, thus, there are no algorithms computing sat. To deal with this issue 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 $(\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge \neg\, nsat(\psi \wedge \phi_1 \wedge \phi_2) \Rightarrow \pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2)$ which is what we actually implemented in our prototype tool in $\mathbb{K}$ Framework. The predicate $nsat$ is an approximation of the theoretical unsatisfiability predicate.

6.1 Adding Symbolic Steps expressing Implication

In the previous section we have shown how to encode the symbolic transition relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$, which is expressed using symbolic rewriting, by using standard rewriting with a modified set of rules $\mathcal{S}^{\mathfrak{s}}$. We naturally want to implement its extension to the verification of RL formulas by standard rewriting as well. Now, verifying RL formulas amounts to computing the transitions between RL formulas generated by the deductive system defined in Figure 5. We note that the set of symbolic rules defined in the previous section encodes almost such transitions, because the derivative of a pattern with respect to a set of rules can be computed by standard rewriting with the symbolic version of the rules. The missing piece is the [Implication] inference rule, for which there is currently no rule in $\mathcal{S}^{\mathfrak{s}}$ that generates a transition $\varphi_1 \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \varphi_2$ when $\models \varphi_1 \rightarrow \varphi_2$ holds. We now show how to enrich the set of rules $\mathcal{S}^{\mathfrak{s}}$ to deal with this case.

Since it is not efficient to add symbolic steps corresponding to all the implications, we have implemented only particular cases, which are the most useful in practice. These correspond to implications of the form $\pi \wedge \phi \longrightarrow (\exists var(\pi'))\pi' \wedge \phi'$, where $(\exists var(\pi')).\pi' \wedge \phi'$ is also the right-hand side of a RL formula denoting the program's specification[2], and $\pi \wedge \phi$ is a pattern reached at some point during a certain proof branch. Then, proving $\models \pi \wedge \phi \longrightarrow (\exists var(\pi')).\pi' \wedge \phi'$ allows our RL verification to terminate that proof branch.

For this, we enrich the set of predicates $\Pi^{\mathfrak{s}}$ with a binary predicate $\_ \vdash \_$ of type $Cond \times Cond$ such that $\mathcal{T}^{\mathfrak{s}}_{\_\vdash\_} \subseteq \{(\phi_1, \phi_2) \mid \models \phi_1 \longrightarrow \phi_2\}$, and for each pattern of interest, of the form

[2]Remember that, for such rules, some variables in π' may be existentially quantified. Without restriction of generality, we can assume that all variables in $var(\pi')$ are quantified, possibly after adding additional variables and equality constraints to ϕ'.

$(\exists var(\pi'))\pi' \wedge \phi'$, we add to $\mathcal{S}^{\mathfrak{s}}$ the rule

$$(\pi' \wedge^{\mathfrak{s}} \psi) \wedge (\psi \vdash \phi') \Rightarrow \pi' \wedge^{\mathfrak{s}} \phi' \tag{4}$$

Similarly to *nsat*, the predicate $\_ \vdash \_$ can be encoded in a theorem prover, not necessarily the same as the one encoding *nsat* (which is Z3 [19]).

Proposition 2 *Assume there is a transition $\pi \wedge^{\mathfrak{s}} \phi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \pi \wedge^{\mathfrak{s}} \phi''$ generated by the rule (4). Then, the validity $\models \pi \wedge \phi \longrightarrow (\exists var(\pi'))\pi' \wedge \phi'$ holds.*

Proof If the transition $\pi \wedge^{\mathfrak{s}} \phi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$ is generated by the rule (4) then there exists $\rho^{\mathfrak{s}} : Var \cup \{\psi\} \to \mathcal{T}^{\mathfrak{s}}$ such that $(\pi' \wedge^{\mathfrak{s}} \psi)\rho^{\mathfrak{s}} = \pi \wedge^{\mathfrak{s}} \phi$, $\rho^{\mathfrak{s}} \models (\psi \vdash \phi')$, and $(\pi' \wedge^{\mathfrak{s}} \phi')\rho^{\mathfrak{s}} = \pi' \wedge^{\mathfrak{s}} \phi''$. Thus, $\pi'\rho^{\mathfrak{s}} = \pi$, $\phi'\rho^{\mathfrak{s}} = \phi''$, $\psi\rho^{\mathfrak{s}} = \phi$, $\models \phi \to \phi''$. There remains to prove $\models \pi \wedge \phi \longrightarrow (\exists var(\pi))\pi \wedge \phi'$. For this, assume $(\gamma, \rho) \models \pi \wedge \phi$, thus, $\gamma = \pi\rho$ and $\rho \models \phi$. Let ρ' be defined such that $x\rho' = x\rho$ for all $x \notin var(\pi')$ and $y\rho' = y\rho^{\mathfrak{s}}$ for all $y \in var(\pi')$. From $\models \phi \to \phi'\rho^{\mathfrak{s}}$ we get $\rho \models \phi'\rho^{\mathfrak{s}}$. This is equivalent to $\rho' \models \phi'$. Thus $(\gamma, \rho') \models \pi' \wedge \phi'$ and the conclusion follows.
Example Consider the right-hand side of the RL formula (3) used for proving the `gcd` program. The instance of the rule (4) generated for it is

$$\begin{array}{l}(\langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a}\mapsto l_1\ \mathtt{b}\mapsto l_2\ \mathtt{x}\mapsto l_3\ \mathtt{y}\mapsto l_4\ \cdots\rangle_{\mathsf{env}}\langle l_1\mapsto a\, l_2\mapsto b\, l_3\mapsto x'\, l_4\mapsto y'\ \cdots\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathrm{cfg}} \wedge^{\mathfrak{s}} \psi) \wedge \\ \qquad (\psi \vdash \phi') \Rightarrow \\ \langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a}\mapsto l_1\ \mathtt{b}\mapsto l_2\ \mathtt{x}\mapsto l_3\ \mathtt{y}\mapsto l_4\ \cdots\rangle_{\mathsf{env}}\langle l_1\mapsto a\, l_2\mapsto b\, l_3\mapsto x'\, l_4\mapsto y'\ \cdots\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathrm{cfg}} \wedge^{\mathfrak{s}} \phi'\end{array}$$

where $\phi' \triangleq gcd(x', y') = gcd(a, b) \wedge x' \geq 0 \wedge y' \geq 0$. The predicate $\_ \vdash \_$ includes the following problem-specific property of the mathematical function *gcd*:

$$(gcd(x, y) = d \wedge y > 0) \vdash gcd(y, x \,\%\, y) = d \tag{5}$$

Using (4) and (5), we obtain the following symbolic step:

$$\begin{array}{l}\langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a}\mapsto l_1\ \mathtt{b}\mapsto l_2\ \mathtt{x}\mapsto l_3\ \mathtt{y}\mapsto l_4\ \cdots\rangle_{\mathsf{env}}\langle l_1\mapsto a\, l_2\mapsto b\, l_3\mapsto y\, l_4\mapsto x \,\%\, y\ \cdots\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathrm{cfg}} \wedge^{\mathfrak{s}} \phi_1 \\ \qquad \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \\ \langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a}\mapsto l_1\ \mathtt{b}\mapsto l_2\ \mathtt{x}\mapsto l_3\ \mathtt{y}\mapsto l_4\ \cdots\rangle_{\mathsf{env}}\langle l_1\mapsto a\, l_2\mapsto b\, l_3\mapsto y\, l_4\mapsto x \,\%\, y\ \cdots\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathrm{cfg}} \wedge^{\mathfrak{s}} \phi_2\end{array}$$

where ϕ_1 denotes the formula $gcd(x, y) = gcd(a, b) \wedge x \geq 0 \wedge y > 0$ and ϕ_2 the formula $gcd(y, x \,\%\, y) = gcd(a, b) \wedge y \geq 0 \wedge x \,\%\, y \geq 0$. The left-hand side pattern of the above $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ transition is obtained as a result of symbolically executing the left-hand side of (3), and the pattern obtained after the $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ transition is an instance of the right-hand side of (3) obtained by the substitution $x' \mapsto y$, $y' \mapsto x \,\%\, y$. Thus, the $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ transition enabled terminating the proof of the formula (3), which was its expected effect. □

7 Implementation in the $\mathbb{K}$ Framework

In this section we present a prototype tool implementing our symbolic execution approach. We first briefly describe the tool and its integration within the $\mathbb{K}$ framework. Then we illustrate the tool (as well as its extension that performs deductive verification of RL formulas) on some nontrivial programs.

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

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

     init(a, n, x, j);
}
```

Figure 6: CinK program: `init-arrays`

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

We have integrated our symbolic execution framework in the $\mathbb{K}$ framework [37]. In $\mathbb{K}$, the definition of a language, say, $\mathcal{L}$, is compiled into a Maude [14] rewrite theory. Then, the $\mathbb{K}$ runner executes programs in $\mathcal{L}$ by applying the resulting rewrite rules to configurations containing programs. Our implementation follows the same process. The main difference is that our new $\mathbb{K}$ compiler includes some additional transformations steps: rule linearisation, replacing the data subterms in left-hand-sides of rules with fresh variables and adding constraints to the rule's condition equating the fresh variables to the terms they replaced, adding a cell for path condition, and modifying rules $\pi \wedge \phi$ into symbolic rules of the form $(\pi_1 \wedge^{\mathbb{s}} \psi) \wedge \neg\, nsat(\psi \wedge \phi_1 \wedge \phi_2) \Rightarrow \pi_2 \wedge^{\mathbb{s}} (\psi \wedge \phi_1 \wedge \phi_2)$ as shown in the previous section. The effect is that the compiled rewrite theory we obtain defines the so-called symbolic semantics of $\mathcal{L}$ instead of its concrete semantics. We note that the symbolic semantics can execute programs with concrete inputs as well. For user convenience we have also improved the $\mathbb{K}$ runtime environment with some specific options which are useful for providing programs with symbolic input and setting up an initial path condition. The predicate *nsat* is implemented by the interaction with the Z3 SMT solver [19]. A more detailed description of our tool and more examples can be found in [6].

Given a $\mathbb{K}$ language definition, our tool automatically generates its symbolic semantics. Thus, users that already have a $\mathbb{K}$ language definition can symbolically execute their programs without having to change anything in them.

7.2 Bounded model checking

We illustrate symbolic execution with CinK and show how the $\mathbb{K}$ runner can directly be used for performing bounded model checking. In the program in Figure 6, the function `init` 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 `i`-th array element is accessed using the pointer `p`. The function `init` is called in the function `main` with arguments read from the standard input. In [5] it has been shown, using model-checking and abstractions on arrays, that this program never prints *error*. It is worth noting that the CinK program used here is trickier than the one in [5] since it uses conversions between arrays and pointers. We obtain the same result as [5] by running the program with symbolic inputs and using the $\mathbb{K}$ runner as a bounded model checker:

```
$ krun init-arrays.cink -cPC="n >Int 0" -search -cIN="n j x a1 a2 a3"
```

```
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

```
                    -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, but it can be specified by the initial path condition as well, e.g., $n <_{Int} 4$. 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 size is symbolic as well, a feature that, to our best knowledge, is not present in other symbolic execution frameworks.

7.3 KOOL: testing virtual method calls on lists

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

The `List` class implements (plain) lists. It has methods for creating, copying, and testing the equality of lists, as well as for inserting and deleting elements in a list. Figure 7 shows only a part of them. The class `OrderedList` inherits from `List`. It redefines the `insert` method in order to ensure that the sequences of elements in lists are sorted in increasing order. The `Main` class creates a list `l1`, initializes `l1` and an integer variable `x` with input values, copies `l1` to a list `l2`

and then inserts and deletes `x` in `l1`. Finally it compares `l1` to `l2` element by element, and prints *error* if it finds them different. We use symbolic execution to show that the above sequence of method calls results in different outcomes, depending on whether `l1` is a `List` or an `OrderedList`. We first try the case where `l1` is a `List`, by issuing the following command to the $\mathbb{K}$ runner:

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

7.4 Reachability-Logic Verification

In this section we present an implementation of the deductive system that we have defined in Figure 5 and illustrate it on the Knuth-Morris-Pratt [28] string matching algorithm. The current implementation is an extension of both our $\mathbb{K}$ symbolic compiler and the $\mathbb{K}$ runner. In order to verify $\mathcal{S} \models G$, for a given language semantics $\mathcal{S}$ and a set of reachability formulas (goals) G, the tool completes two stages during its execution: it builds a new definition and then performs verification. Given a language definition $\mathcal{L}$ and a set of RL formulas G, the tool produces a new definition consisting of the symbolic semantics $\mathcal{L}^{\mathbf{s}}$ of $\mathcal{L}$ enriched with the rewrite rules from G. This new definition is used to perform symbolic execution of the patterns in left-hand sides of formulas in $\Delta_{\mathcal{S}}(G)$, in order to find a proof of $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ with the proof system in Fig. 5. Computing all the successors of a pattern by applying rules from the symbolic semantics of $\mathcal{L}$ corresponds to applying the SymbolicStep deduction rule. Computing the successor of a pattern by using a rule from G corresponds to applying CircularHypothesis. The rules from G are only applied to patterns from the left-hand side of $\Delta_{\mathcal{S}}(G)$ or to their successors obtained by derivation. This is achieved using a tagging mechanism that singles out the patterns to which rules from G can be applied, and, moreover, this gives priority to rules in G over rules in the symbolic semantics. CaseAnalysis is implicitly applied by splitting disjunctive patterns $\bigvee_i(\pi_i \wedge \phi_i)$ obtained by SymbolicStep and CircularHypothesis back into elementary patterns $\pi_i \wedge \phi_i$. Implication is applied at the end of the branches of the proof tree, in order to check that the current pattern implies the right-hand side of the current goal. This has to succeed in all branches for the proof to

succeed. Transitivity is used to build proof trees in the deductive system in Fig. 5 from individual proof steps.

For instance, the partial correctness of `gcd.cink` (Figure 4) is proved using G given by the RL rules (1)-(3). The tool has also been used to prove all the programs from [4] written in a simple imperative language IMP using the definition of IMP included in the $\mathbb{K}$ release. In the rest of the section we illustrate the verification of a CinK implementation of the Knuth-Morris-Pratt string matching algorithm, to show that our language-independent approach can deal with nontrivial proofs.

7.4.1 Verifying the Knuth-Morris-Pratt string matching algorithm: `KMP`

The Knuth-Morris-Pratt algorithm [28] searches for occurrences of a word P, usually called *pattern*, within a given text T by making use of the fact that when a mismatch occurs, the pattern contains sufficient information to determine where the next search should begin. A detailed description of the algorithm, whose CinK code is shown in Figure 8, can be found in [16].

The `KMP` algorithm optimises the naive search of a pattern into a given string by using some additional information collected from the pattern. For instance, let us consider $T =$ `ABADABCDA` and $P =$ `ABAC`. It can be easily observed that `ABAC` does not match `ABADABCDA` starting with the first position because there is a mismatch on the fourth position, namely `C` $\neq$ `D`.

The `KMP` algorithm uses a *failure function* π, which, for each position j in P, returns the length of the longest proper prefix of the pattern which is also a suffix of it. For our example, $\pi[3] = 1$ and $\pi[j] = 0$ for $j = 1, 2, 4$. In the case of a mismatch between the position i in T and the position j in P, the algorithm proceeds with the comparison of the positions i and $\pi[j]$. For the above mismatch, the next comparison is between the `B` in `ABAC` and the first instance of `D ABADABCDA`, which saves a comparison of the characters preceeding them, since the algorithm "already knows" that they are equal (here, they are both `A`).

An implementation of `KMP` is shown in Figure 8. The comments include the specifications for preconditions, postconditions, and invariants, which will be explained later in this section (briefly, they are syntactic sugar for RL formulas, which are automatically generated from them). The program can be run either using the $\mathbb{K}$ semantics of CinK or the `g++` GNU compiler. The `compute_prefix` function computes the failure function π for each component of the pattern and stores it in a table, called `pi`. The `kmp_matcher` searches for all occurrences of the pattern in the string comparing characters one by one; when a mismatch is found on positions i in the string and q in the pattern, the algorithm shifts the search to the right as many positions as indicated by `pi[`q`]`, and initiates a new search. The algorithm stops when the string is completely traversed.

For the proof of `KMP` we use the original algorithm as presented in [16]. Another formal proof of the algorithm is given in [21] by using Why3 [22]. There, the authors collapsed the nested loops into a single one in order to reduce the number of invariants they have to provide. They also modified the algorithm to stop when the first occurrence of the pattern in the string was found. By contrast, we do not modify the algorithm from [16]. We also prove that `KMP` finds *all* the occurrences of the pattern in the string, not only the first one. We let $P[1..i]$ denote the prefix of P of size i, and $P[i]$ denote its i-th element.

Definition 16 *Let P be a pattern of size $m \geq 1$ and T a string of characters of size $n \geq 1$. We define the following functions and predicate:*

- *$\pi(i)$ is the length of the longest proper prefix of $P[1..i]$ which is also a suffix for $P[1..i]$, for all $1 \leq i \leq m$;*

```
/*@pre: m>=1 */
void compute_prefix(char p[],
                    int m, int pi[])
{
  int k, q;
  k = 0;
  pi[1] = 0;
  q = 2;
  while(q <= m) {
  /*@inv: 0<=k /\ k<q /\ q<=m+1 /\
    (forall u:1..k)(p[u]=p[q-k+u]) /\
    (forall u:1..q-1)(pi[u]=Pi(u)) /\
    Pi(q)<=k+1 */
    while (k > 0 && p[k+1] != p[q]) {
    /*@inv: 0<=k /\ k<q /\ q<=m /\
      (forall u:1..k)(p[u]=p[q-k+u]) /\
      (forall u:1..q-1)(pi[u]=Pi(u)) /\
      (forall u:1..m)(0<=Pi(u)<u) /\
      Pi(q)<=k+1  */
      k = pi[k];
    }
    if (p[k + 1] == p[q]) {
      k = k + 1;
    }
    pi[q] = k;
    q++;
  }
}
/*@post: (forall u:1..m)(pi[u]=Pi(u)) */
```

```
/*@pre: m>=1 /\ n>=1 */
void kmp_matcher(char p[], char t[], int m, int n)
{
  int q = 0, i = 1, pi[m];
  compute_prefix(p, m, pi);
  while (i <= n) {
  /*@inv: 1<=m /\ 0<=q<=m /\ 1<=i<=n+1 /\
    (forall u:1..q-1)(pi[u]=Pi(u)) /\
    (exists v)(forall u:v+1..i-1)(Theta(u)<m /\
                          allOcc(Out,p,t,v))/\
    (forall u:1..q)(p[u]=t[i-1-q+u]) /\
    Theta(i)<=q+1 */
    while (q > 0 && p[q + 1] != t[i]) {
    /*@inv: 1<=m /\ 0<=q /\ q<m /\
    (forall u:1..q-1)(pi[u]=Pi(u)) /\
    (exists v)(forall u:v+1..i-1)(Theta(u)<m /\
                          allOcc(Out,p,t,v))/\
    (forall u:1..q)(p[u]=t[i-1-q+u]) /\
    (forall u:1..i-1)(Theta(u)<m) /\
    Theta(i)<=q+1 */
      q = pi[q];
    }
    if (p[q + 1] == t[i]) { q = q + 1;  }
    if (q == m) {
      cout << "shift: " << (i - m) << endl;
      q = pi[q];
    }
    i++;
  }}
/*@post: allOcc(Out, p, t, n) */
```

Figure 8: The `KMP` algorithm annotated with pre-/post-conditions and invariants: failure function (left) and the main function (right). Note that we used `Pi`, `Theta`, and `allOcc` to denote functions π and θ, and predicate *allOcc*, respectively.

- $\theta(i)$ *is the length of the longest prefix of* P *that matches* T *on the final position* i*, for all* $1 \leq i \leq n$*;*
- $allOcc(Out, P, T, i)$ *holds iff the list* Out *contains all the occurrences of* P *in* $T[1..i]$.

The specification of the `kmp_matcher` function is the following RL formula:

$$\left\langle \begin{array}{l} \langle \texttt{kmp\_matcher(p, t,}m\texttt{,}n\texttt{);} \rangle_{\mathsf{k}} \langle \cdot \rangle_{\mathsf{out}} \\ \langle \texttt{p} \mapsto l_1 \ \texttt{t} \mapsto l_2 \rangle_{\mathsf{env}} \langle l_1 \mapsto P \ l_2 \mapsto T \rangle_{\mathsf{store}} \end{array} \ \ldots \right\rangle_{\mathsf{cfg}} \wedge n \geq 1 \wedge m \geq 1$$

$$\Rightarrow$$

$$\langle \langle \cdot \rangle_{\mathsf{k}} \langle Out \rangle_{\mathsf{out}} \langle \ldots \rangle_{\mathsf{env}} \langle \ldots \rangle_{\mathsf{store}} \ldots \rangle_{\mathsf{cfg}} \wedge \quad allOcc(Out, P, T, n)$$

This formula says that from a configuration where the program variables `p` and `t` are bound to the values P, T, respectively, the output cell is empty, and the `kmp_matcher` function has to be executed, one reaches a configuration where the function has been executed and the output cell contains all the occurrences of P in T. Note that we passed the symbolic values m and n as actual parameters to the function which are the sizes of P, and T, respectively. An advantage of RL with respect to Hoare Logic is, in addition to language independence, the fact that RL formulas may refer to all the language's configuration, whereas Hoare Logic formulas may only refer to program variables. A Hoare Logic formula for the `kmp_matcher` function would require the addition of assignments to a new variable playing the role of our output cell.

There are some additional issues concerning the way users write the RL formulas. These may be quite large depending on the size of the $\mathbb{K}$ configuration of the language. To handle that, we have created an interactive tool for generating such formulas. Users can annotate their programs with preconditions and postconditions and then use our tool to generate RL formulas from those annotations. The above specification for `KMP` is generated from the annotations:

```
//@pre: m >= 1 /\ n >= 1
kmp_matcher(p, t, m, n);
//@post: allOcc(Out, p, t, n)
```

Loops can be annotated with invariants as shown below:

```
while (COND) {
  //@inv: INV
  k = pi[k];
}
```

For each annotated loop, the tool generates two RL formulas: one for proving the loop body and another one for proving the entire loop statement. The former states that, starting from a configuration where the body of the loop remains to be executed (e.g. `k = pi[k]` in the above loop) and the FOL formula `INV` $\wedge$ `COND` holds, one reaches a configuration where the body was executed and `INV` holds. The latter states that by starting with a configuration where the entire loop remains to be executed and `INV` holds, one reaches a configuration where the loop was completely executed and `INV` $\wedge \neg$`COND` holds.

From the annotations shown in Figure 8 the tool generates all the RL formulas that we need to prove `KMP`. Since `KMP` has four loops and two pairs of pre/post-conditions, the tool generates and proves a total number of ten RL formulas. In the annotations we use the program variables (e.g. `pi`, `p`, `m`) and a special variable `Out` which is meant to refer the content of the $\langle \rangle_{\mathsf{out}}$ cell. This variable gives us access to the output cell, which is essential in proving that the algorithm computes all the occurrences of the pattern.

Finally, as already noted in Section 6.1, every particular verification problem requires problem-specific constructions and properties about them. For verifying `KMP` we have enriched the symbolic definition of CinK with functional symbols for π, θ, and $allOcc$, and the following facts about the $\vdash$ entailment, expressing some of their properties (which we prove independently in Coq [1]):

1. $0 \leq k \leq m \vdash 0 \leq \pi(k) < k$.
2. $0 \leq q \leq n \vdash 0 \leq \theta(q) \leq m$.
3. $(\forall u : 1..k)(P[u] = P[q - k + u]) \wedge \pi(q) \leq k + 1 \wedge P[k + 1] \neq P[q] \vdash$ $\pi(q) \leq \pi(k) + 1$.
4. $(\forall u : 1..k)(P[u] = P[q - k + u]) \wedge \pi(q) \leq k + 1 \wedge P[k + 1] = P[q] \vdash$ $\pi(i) = k + 1$.
5. $(\forall u : 1..q)(P[u] = T[i - 1 - q + u]) \wedge \theta(i) \leq q + 1 \wedge P[q + 1] \neq T[i] \vdash$ $\theta(i) \leq \pi(q) + 1$.
6. $(\forall u : 1..q)(P[u] = T[i - 1 - q + u]) \wedge \theta(i) \leq q + 1 \wedge P[q + 1] = T[i] \vdash$ $\theta(i) = q + 1$.
7. $(\exists v)(\forall u : v{+}1..i{-}1)(allOcc(Out, P, T, v) \wedge \theta(u) < m) \wedge \theta(i) = m \wedge i < n \vdash$ $(\exists v)(\forall u : v{+}1..i)(allOcc(Out, P, T, v) \wedge \theta(u) < m)$.
8. $(\exists v)(\forall u : v{+}1..i)(allOcc(Out, P, T, v) \wedge \theta(u) < m) \wedge i = n \vdash$ $allOcc(Out, P, T, v)$.

8 Conclusion and Future Work

We have presented a formal and generic framework for the symbolic execution of programs in languages definable in an algebraic and term-rewriting setting. Symbolic execution is performed by applying the rules of a language's semantics by so-called symbolic rewriting. We prove that the symbolic execution thus defined has the naturally expected properties with respect to concrete execution: *coverage*, meaning that to each concrete execution there is a feasible symbolic on on the same path of instructions, and *precision*, meaning that each feasible symbolic execution has a concrete execution on the same path. These properties are expressed in terms of mutual simulations. The incorporation of symbolic execution into a deductive system for program verification with respect to Reachability-Logic specifications, developed in detail elsewhere, is also briefly presented. In order to implement our symbolic rewriting-based approach in a setting where only standard execution is available, such as the $\mathbb{K}$ framework, we define a transformation of language definitions $\mathcal{L}$ into other language definitions $\mathcal{L}^{\mathfrak{s}}$, and show that concrete program execution $\mathcal{L}^{\mathfrak{s}}$, which uses standard rewriting, are in a covering&precise relationship with the symbolic execution of the corresponding programs in $\mathcal{L}$. Finally, we present the implementation of a prototype tool based on the above theory, which is now a part of the $\mathbb{K}$ framework, and its applications to the bounded model checking and deductive verification of nontrivial programs written in a subset of C++ also formally defined in $\mathbb{K}$.

Future Work We are planning to expand our tool, to make it able to seamlessly perform a wide range of program analyses, from testing and debugging to formal verifications, following ideas presented in related work, but with the added value of being language independent and grounded in formal methods. For this, we shall develop a rich domain of symbolic values, able to handle

various kinds of data types. Formalising the interaction of symbolic-domain computations with symbolic execution is also a matter for future work.

Another future research direction is specifically targeted at our RL-formulas verifier, and aims at certifying its executions. The idea is to generate proof scripts for the Coq proof assistant [1], in order to obtain certificates that, despite any (inevitable) bugs in our tool, the proofs it generates are indeed correct. This amounts to, firstly, encoding our RL proof system in Coq, and proving its soundness with respect to the original proof system of RL (which have already been proved sound in Coq [36]). Secondly, our verifier must be enhanced to return, for any successful execution, the rules of our system it has applied and the substitutions it has used. From this information a Coq script is built that, if successfully run by Coq, generates a proof term that constitutes a correctness certificate for the verifier's original execution. A longer-term objective is to turn our verifier into an external proof tactic for Coq, resulting in a powerful mixed interactive/automatic program verification tool.

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] The Coq proof assistant reference manual, `http://coq.inria.fr/refman/`.

[2] Standard for Programming Language C++. Working Draft. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

[3] W. Ahrendt. The KeY tool. *Software and Systems Modeling*, 4:32–54, 2005.

[4] K. R. Apt, F. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer Verlag, 3rd edition, 2009.

[5] A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. *Electr. Notes Theor. Comput. Sci.*, 144(3):79–94, 2006.

[6] 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/`.

[7] A. Arusoaie, D. Lucanu, and V. Rusu. Language-Independent Program Verification Using Symbolic Execution. Rapport de recherche RR-8369, INRIA, Sept. 2013.

[8] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In *Proc. 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, 2005.

[9] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In Yi [42], pages 52–68.

[10] D. Bogdănaş. Java semantics in $\mathbb{K}$. `https://github.com/kframework/java-semantics`.

[11] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, 2008.

[12] 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.

[13] E. Clarke and D. Kroening. Hardware verification using ansi-c programs as a reference. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 308–311, New York, NY, USA, 2003. ACM.

[14] M. Clavel, F. Dur'an, S. Eker, P. Lincoln, N. M. Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science. Springer, July 2007.

[15] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Softw. Eng. Notes*, 26(5):142–151, 2001.

[16] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[17] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. *SIGPLAN Not.*, 48(4):329–342, Mar. 2013.

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

[19] 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.

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

[21] J. C. Filliâtre. Proof of kmp string searching algorithm. `http://toccata.lri.fr/gallery/kmp.en.html`.

[22] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[23] 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.

[24] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.

[25] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Tracer: a symbolic execution tool for verification. In *Proc. 24th international conference on Computer Aided Verification*, CAV'12, pages 758–766. Springer-Verlag, 2012.

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

[28] D. E. Knuth, J. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[29] D. Lucanu and T. F. Serbanuta. CinK - an exercise on how to think in K. Technical Report TR 12-03, Version 2, Alexandru Ioan Cuza University, Faculty of Computer Science, December 2013.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[36] 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.

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

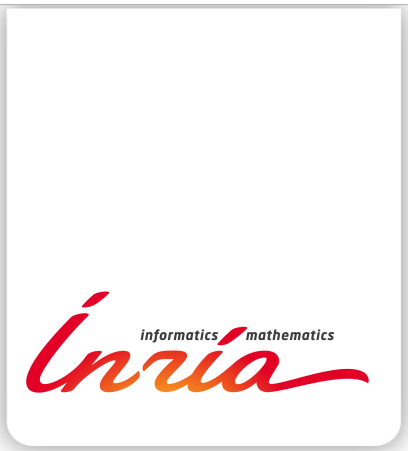
[38] G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 555–574. ACM, 2012. Also available as technical report `http://hdl.handle.net/2142/33771`.

[39] 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.

[40] 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.

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

[42] 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.



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