



HAL
open science

A Generic Framework for Symbolic Execution: a Coinductive Approach

Dorel Lucanu, Vlad Rusu, Andrei Arusoaie

► **To cite this version:**

Dorel Lucanu, Vlad Rusu, Andrei Arusoaie. A Generic Framework for Symbolic Execution: a Coinductive Approach. *Journal of Symbolic Computation*, 2017, 80, pp.125-163. 10.1016/j.jsc.2016.07.012 . hal-01238696v2

HAL Id: hal-01238696

<https://inria.hal.science/hal-01238696v2>

Submitted on 6 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Generic Framework for Symbolic Execution: Theory and Applications

Dorel Lucanu^a Vlad Rusu^b Andrei Arusoaie^{a,b}

^a*Faculty of Computer Science, “Alexandru Ioan Cuza” University of Iași, Romania*
{andrei.arusoaie,dlucanu}@info.uaic.ro

^b*Inria Lille Nord Europe, France*
vlad.rusu@inria.fr

Abstract

We propose a language-independent symbolic execution framework. The approach is parameterised by a language definition, which consists of a signature for the syntax and execution infrastructure of the language, a model interpreting the signature, and rewrite rules for the language’s operational semantics. Then, symbolic execution amounts to computing symbolic paths using a *derivative* operation. 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 also show how a coinduction-based extension of symbolic execution can be used for the deductive verification of programs. We show how the proposed symbolic-execution approach, and the coinductive verification technique based on it, can be seamlessly implemented in language definition frameworks based on 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 analysis and deductive verification of nontrivial programs.

Key words: symbolic execution, programming language, formal operational semantics, reachability logic, circular coinduction, program verification

1 Introduction

Symbolic execution is a well-known program analysis technique introduced in 1976 by James C. King [24]. 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 [34]. The main advantage of symbolic execution is that it allows to reason 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.

In this paper we address two foundational issues regarding the symbolic execution, namely its relationships with the formal definition of the language, for soundness, and with the program logics, for applications to program analysis and verification.

Description of the 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 rewriting¹. On the theoretical side, we define symbolic execution as the application of rewrite rules in the semantics by *derivation*, a logical description of symbolic successors of a given set of states also symbolically represented as a logical formula in Matching Logic (ML) [38]. We prove that the symbolic execution thus defined has properties ensuring 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.

We also show how a simple extension of our symbolic-execution approach results in a deductive system for proving programs with respect to Reachability Logic (RL) [13] properties; RL is a language-independent program logic also used for defining language semantics, which has been shown to subsume existing language-dependent logics such as Hoare and Separation logics [39,43]. The proposed deductive system is proved to be sound by using a coinductive proof technique. It is shown to be a strict generalization of an approach we presented in [26], in the sense that the procedure for RL proposed there is a strategy of the proof system proposed here. Our 3-rule proof system is also substantially simpler than the original 8-rule proof system given in [13]; the price to pay is the theoretical relative completeness property, which the original proof system has, whereas ours is not known to have. The proof system we propose is inspired from the circular coinduction proof technique [41], applied in this paper to programming language definitions (whereas in [41] it is applied to proving observational equalities between possibly infinite data structures,

¹ Most existing operational semantics styles (small-step, big-step, reduction with evaluation contexts, ...) have been shown to be representable in this way in [48].

e.g., streams). This was possible by defining an appropriate notion of derivative in the new context and by exploiting the common framework of induction and coinduction based on ground rules [45]. We thus obtain a uniform and rigorous approach for both finite and infinite symbolic executions.

On the practical side, we present an implementation of the theory in a prototype implementation based on \mathbb{K} [42], a framework dedicated to defining formal operational semantics of languages. Our current prototype is built on version 3.4 of \mathbb{K} (<https://github.com/kframework/k/releases/tag/v3.4>) and enhances the previous one based on the language transformation [4]. \mathbb{K} is based on rewriting, hence, we formally prove that the derivation operation can be correctly implemented by applying certain modified rewrite rules (obtained by automatically transforming the original ones) over ML formulas. This additional intermediary step between abstract theory and implementation is important for ensuring that the resulting prototype tool adequately implements the theory, since the two extreme sides of our approach lie at quite distant levels of abstraction. We describe our prototype and demonstrate it on examples, which illustrate the bounded model checking of programs and their deductive verification with respect to RL formulas.

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.

An approach closely related to ours is implemented in the MatchC tool [40,43], which has been used for verifying challenging C programs such as the Schorr-Waite garbage collector. MatchC also uses the formalism of Reachability Logic for program specifications; the MatchC tool implementation, is, however, dedicated to a subset of C. The main difference with our approach is that we emphasise on *bridging the gap* between theory and implementation (from an initial, abstract definition of symbolic execution to its extension for program verification, then to its encoding by rewriting and finally to its implementation in code), whereas in the MatchC tool it does not focus on intermediary steps between theory (a language-independent deductive system) and code implementing it for a subset of C. The same comparison holds between our approach and [13], in which a deductive system for a different version of RL is implemented for verifying programs written in a specific language. Since our approach of symbolic execution is founded on coinduction, it can also be seen as a bridge between the pure coinductive program verification techniques [30] and verification techniques based on operational semantics [13,40].

Java PathFinder [32] 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 between multiple decision procedures. The tool is fully dedicated to the Java language.

Another approach consists of 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 [20], CUTE [47], EXE [8], PEX [15]. 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 [7], while others are more oriented towards verification [11,23,35]. Several techniques are implemented to improve the performance of these tools, such as *bounded verification* [9] and *pruning* the execution tree by eliminating redundant paths [14]. Other approaches offer support for verification of code contracts over programs. Spec# [5] 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 [1] tool. In particular, KeY allows to prove that after running a method, its postcondition and the class invariant holds, using Dynamic Logic [21] and symbolic execution. The VeriFast tool [22] supports verification of single and multi-threaded C and Java programs annotated with preconditions and postconditions written in Separation Logic [36]. The Smallfoot tool [6] uses symbolic execution together with separation logic to prove Hoare triples. There are also approaches that attempt to automatically detect invariants in programs [33,46]. 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).

Regarding performance, 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.

Symbolic execution in term-rewriting systems is another related work. The technique called *narrowing*, initially used for solving equation systems in ab-

stract datatypes, was extended for solving reachability problems in term-rewriting systems and was applied to the analysis of security protocols [29]. Such analyses rely on powerful unification-modulo-theories algorithms [17], 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.

A recent evolution of symbolic execution in term-rewriting systems is *rewriting modulo SMT* [37], in which terms are constrained with unquantified first-order formulas over *builtin sorts*, and the constraints accumulated during term-rewriting are handled by SMT solving. For non-builtin sorts unification modulo theories reduces (under technical conditions) to matching modulo theories. There are some similarities with our approach, e.g., they also obtain mutual simulations between symbolic and concrete rewriting. However, the unquantified constrained terms that [37] can rewrite over are a subset of ML, hence, our approach, which handles the full ML logic, is more general.

Finally, there is the comparison with our own previous work on this topic. The conference paper [3] presents our initial approach, and has since evolved into several directions. The present paper extends [3] with the following features: more expressive language definitions (including configurations satisfying associativity and/or commutativity and/or unity (ACU) axioms, an essential feature in real languages definition in the \mathbb{K} framework); a more expressive (and thus more precise) symbolic model (ML formulas vs. terms constrained with Boolean formulas); a more general presentation of symbolic execution (using the logical construction of derivatives, for which the approach based on language transformation of [3] is only an implementation); and the RL-formula verification application of symbolic execution. We also systematically employ (co)induction in order to reason about possibly infinite program executions.

The paper [4] expands another idea present in [3], namely, language transformations for symbolic execution, for a different symbolic model (equivalence classes of ML formulas having the same semantics, which is in some sense the best possible model in terms of precision with respect to concrete program configurations). Language transformations techniques to compute for such a symbolic model are proposed in [4]. Moreover, the paper [44] deals with the implementation of concrete and symbolic execution in programming languages using *rewrite theories*; precise and approximated implementations are proposed. While ML formulas as a precise symbolic model, and combination of ACU axioms, are considered in [4,44], neither of those papers deal with RL formula verification, nor do they employ (co)induction as a systematic, rigorous technique for reasoning about possibly infinite program executions. RL formula verification is dealt with in [26], but the procedure for RL verification proposed there is only a particular case of the proof system proposed here.

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: coinduction, a general technique for defining and reasoning about possibly infinite objects such as program executions; reachability logic, which is used for defining operational semantics of languages and for stating program properties; and a generic language-definition framework, in order to make our approach independent of the \mathbb{K} language-definition framework. Section 4 contains our formalisation of symbolic execution, including the coverage and precision results stated earlier in this introduction. Section 5 presents how reachability logic formulas can be verified using a coinductive extension of symbolic execution. In Section 6 we show how symbolic execution and its core derivative operation can be implemented in language definitions based on standard rewriting, such as the \mathbb{K} framework. 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

In this section we present CinK [27], a kernel of the C++ programming language. The \mathbb{K} definition of CinK used here is available on the \mathbb{K} Framework Github repository, <http://github.com/kframework/cink-semantic/releases/tag/v1.0>. 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. The most important is the attribute *strict* that specifies that the arguments of the operator defined by the corresponding production are evaluated in a strict nondeterministic order before evaluating the operator itself. The attribute *strict* may have parameters, which are in fact *sub-attributes*. For instance, the sub-attribute *context*, which can be applied to a specified argument or to *all* arguments, may specify that the corresponding sub-expressions have to be evaluated as rvalues, by wrapping them with the semantic operator *rvalue*. Otherwise, the subexpressions are implicitly evaluated as lvalues (see, e.g., the syntax for the increment and decrement operators). The difference between the two kinds of evaluations can be observed in the rules giving semantics to the operators (Figure 3). Other attributes (*funcall*, *divide*, *plus*, *minus*, ...) are names associated to each syntactic production, which can be used to refer to them.

$$\begin{aligned}
Exp & ::= Id \mid Int \\
& \mid ++ Exp & [strict, prefinc] \\
& \mid -- Exp & [strict, prefdec] \\
& \mid Exp / Exp & [strict(all(context(rvalue))), divide] \\
& \mid Exp + Exp & [strict(all(context(rvalue))), plus] \\
& \mid Exp > Exp & [strict(all(context(rvalue)))] \\
Stmt & ::= Exps ; & [strict] \\
& \mid \{Stmts\} \\
& \mid \text{while } (Exp) Stmt \\
& \mid \text{return } Exp ; & [strict(all(context(rvalue)))] \\
& \mid \text{if } (Exp) Stmt \text{ else } Stmt & [strict(1(context(rvalue)))]
\end{aligned}$$

Fig. 1. Fragment of CinK syntax

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

Fig. 2. CinK configuration structure

Program states are described in \mathbb{K} framework as configurations. A *configuration* is a nested structure of cells, specified using an XML-like notation. The (partial) configuration of CinK (Figure 2) includes the following cells:

- $\langle \cdot \rangle_{cfg}$ – the top cell that includes the other cells;
- $\langle \cdot \rangle_k$ – containing the code that remains to be executed, which is represented as a list of sequential computation tasks $C_1 \curvearrowright C_2 \curvearrowright \dots$. Computation tasks are typically statements and expression evaluations;
- $\langle \cdot \rangle_{env}$ – holding a map from variables to addresses (locations);
- $\langle \cdot \rangle_{store}$ – holding a map from addresses to values);
- $\langle \cdot \rangle_{stack}$ – storing function call stack;
- $\langle \cdot \rangle_{return}$ – for the return values of functions; and
- $\langle \cdot \rangle_{out}$ – holding the 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 $\langle \cdot \rangle_k$ cell and all the other cells remain empty. This is specified by the special variable $\$PGM$ in the cell $\langle \cdot \rangle_k$ and the dot in the other cells.

The semantics of the syntactical constructs is given by \mathbb{K} rules. 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

$I_1 + I_2 \Rightarrow I_1 +_{Int} I_2$	$[plus]$
$I_1 / I_2 \wedge I_2 \neq_{Int} 0 \Rightarrow I_1 /_{Int} I_2$	$[division]$
$\langle ++lval(L) \Rightarrow lval(L) \dots \rangle_k \langle \dots L \mapsto (V \Rightarrow V +_{Int} 1) \dots \rangle_{store}$	$[inc]$
$\langle --lval(L) \Rightarrow lval(L) \dots \rangle_k \langle \dots L \mapsto (V \Rightarrow V -_{Int} 1) \dots \rangle_{store}$	$[dec]$
$\langle \langle lval(L) = V \Rightarrow V \dots \rangle_k \langle \dots L \mapsto _ \Rightarrow V \dots \rangle_{store} \dots \rangle_{cfg}$	$[update]$
$if(true) St \text{ else } _ \Rightarrow St$	$[if-true]$
$if(false) _ \text{ else } St \Rightarrow St$	$[if-false]$
$while(B) St \Rightarrow if(B) \{ St \text{ while}(B) St \text{ else } \{\} \}$	$[while]$
$V ; \Rightarrow \cdot$	$[instr-expr]$
$\{ Sts \} \Rightarrow Sts$	$[block]$
$\langle X \Rightarrow lval(L) \dots \rangle_k \langle \dots X \mapsto L \dots \rangle_{env}$	$[lvalue]$
$rvalue(lval(L : Loc) \Rightarrow \$lookup(L))$	$[rvalue]$
$\langle \langle \$lookup(L) \Rightarrow V \dots \rangle_k \langle \dots L \mapsto V \dots \rangle_{store} \dots \rangle_{cfg}$	$[lookup]$

Fig. 3. Subset of rules from the K semantics of CinK

Figure 3, represents the (concrete) rule

$$\begin{aligned}
& \langle \langle I_1 + I_2 \curvearrowright C \rangle_k \langle E \rangle_{env} \langle S \rangle_{store} \langle T \rangle_{stack} \langle V \rangle_{return} \langle O \rangle_{out} \rangle_{cfg} \\
& \Rightarrow \\
& \langle \langle I_1 +_{Int} I_2 \curvearrowright C \rangle_k \langle E \rangle_{env} \langle S \rangle_{store} \langle T \rangle_{stack} \langle V \rangle_{return} \langle O \rangle_{out} \rangle_{cfg}
\end{aligned} \tag{1}$$

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 rules for the other binary arithmetic operators are similar, excepting the rule for division that has a side condition which restricts its application. The increment and update rules have side effects in the $\langle \rangle_{store}$ cell, modifying the value stored at a specific address.

We briefly describe the rules for the statements. The semantics of the conditional statement **if** is given by two corresponding rules, one for each possible evaluation of the condition expression. The rule giving semantics to the **while** loop performs an unrolling into an **if** statement. This unrolling process does not generate infinite rewritings because the rule can be applied only when the statement **while** is on top of the cell k . The rules giving semantics to the expression and block statements are self-explaining.

The last rules evaluate the program variables as lvalues and give semantics to the **rvalue** wrapper introduced by the sub-attribute **context**, respectively. The auxiliary construct **\$lookup** is used is also used in other rules, not included in the paper.

The semantics of the attribute *strict* is given by so-called *heating* and *cooling* rules, which are automatically generated. We show the case of addition:

$$\begin{array}{ll} A_1 + A_2 \Rightarrow rvalue(A_1) \curvearrowright \square + A_2 & rvalue(I_1) \curvearrowright \square + A_2 \Rightarrow I_1 + A_2 \\ A_1 + A_2 \Rightarrow rvalue(A_2) \curvearrowright A_1 + \square & rvalue(I_2) \curvearrowright A_1 + \square \Rightarrow A_1 + I_2 \end{array}$$

where \square is a special symbol, destined to receive the result of an evaluation.

Example 2.1 *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;
}

```

Fig. 4. Sample CinK Program: gcd

3 Background

In this section we present some theoretical material used in the rest of the paper: induction and coinduction (Section 3.1), a general technique for reasoning about finite and possibly infinite objects such as program executions; reachability logic (Section 3.2), which is used for defining operational semantics of languages and for stating program properties; and language definitions (Section 3.3), which captures the essence of language definition frameworks such as \mathbb{K} . We note that Sections 3.1 and 3.2 adapt material from the literature ([45] and [13], respectively) and Section 3.3 also reuses some of our own [3,26].

3.1 (Co)induction

In this section we present some material on coinduction, which we intensively use in the rest of the paper. Most of the material is adapted from [45]. Coinduction is *dual* to induction, a perhaps better-known reasoning technique, which we also use in a few instances hereafter; we thus present them together.

The common framework for induction and coinduction is generally given using lattice theory, but here we consider only the particular case of powerset lattices, where the partial order is given by inclusion. Let \mathcal{U} be a set and $\mathcal{P}(\mathcal{U})$ its powerset (the set of all subsets of \mathcal{U}). A function $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ is *monotone* if $F(X) \subseteq F(Y)$ whenever $X \subseteq Y$, for all $X, Y \in \mathcal{P}(\mathcal{U})$. Given

a function $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$, an element $X \in \mathcal{P}(\mathcal{U})$ is a *fixed point* of F if $F(X) = X$. If X is the least element (or greatest element) in the set of fixed points of F then X is called the *least fixed point* (or *greatest fixed point*, respectively) of F . The following result is an instance of a more general one known as Knaster-Tarski Theorem:

Theorem 3.1 *Let \mathcal{U} be a set and $\mathcal{P}(\mathcal{U})$ its powerset. Any monotone function $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ has a least fixed point $\mu Y.F(Y)$ (abbreviated as μF) and a greatest fixed point $\nu Y.F(Y)$ (abbreviated as νF). Moreover,*

$$\mu F = \bigcap \{X \mid F(X) \subseteq X\}$$

and

$$\nu F = \bigcup \{X \mid X \subseteq F(X)\}.$$

Definition 3.1 *Let \mathcal{U} be a set and $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ be a monotone function. Then μF is the set inductively defined by F and νF is the set coinductively defined by F .*

A function $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ is *continuous* if $F(\bigcup_{n \geq 0} X_n) = \bigcup_{n \geq 0} F(X_n)$ for any increasing chain $X_0 \subseteq X_1 \subseteq \dots$, and F is *cocontinuous* if $F(\bigcap_{n \geq 0} X_n) = \bigcap_{n \geq 0} F(X_n)$ for any decreasing chain $X_0 \supseteq X_1 \supseteq \dots$. The following result, which is an instance of Kleene's Theorem, shows how the least fixed point and the greatest fixed point can be computed when F is (co)continuous.

Theorem 3.2 *If $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ is continuous then $\mu F = \bigcup_{n \geq 0} F^n(\emptyset)$. If $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ is cocontinuous then $\nu F = \bigcap_{n \geq 0} F^n(\mathcal{U})$.*

Theorem 3.1 suffices for the purposes of this paper, Theorem 3.2 being used only for some particular examples. We often use the case when the function $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ is defined by a set of inference rules.

Definition 3.2 *Let \mathcal{U} be a set. A ground inference rule on \mathcal{U} is a pair (S, x) , where $S \subseteq \mathcal{U}$, $x \in \mathcal{U}$. A set \mathcal{R} of ground rules yields a function $\widehat{\mathcal{R}} : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ given by*

$$\widehat{\mathcal{R}}(X) = \{x \mid (\exists S' \subseteq X)(S', x) \in \mathcal{R}\}.$$

If $S = \{x_1, x_2, \dots\}$, then a rule (S, x) is written as

$$\frac{x_1 \quad x_2 \quad \dots}{x}$$

The function $\widehat{\mathcal{R}}$ associated to a set of ground rules \mathcal{R} is monotone. It follows that each set of ground rules \mathcal{R} inductively defines a set $\mu \widehat{\mathcal{R}}$ and coinductively defines a set $\nu \widehat{\mathcal{R}}$.

A particular case when $\widehat{\mathcal{R}}$ is (co)continuous is given by the following result.

Proposition 3.1 *Let \mathcal{R} be a set of ground rules. If for any $(S, x) \in \mathcal{R}$, S is finite, then $\widehat{\mathcal{R}}$ is continuous.*

If for any x , the set $\{S \mid (S, x) \in \mathcal{R}\}$ is finite, then $\widehat{\mathcal{R}}$ is cocontinuous.

Example 3.1 *The system LIST, given below, coinductively defines the possibly infinite lists over integers:*

$$[A] \frac{}{nil} \quad [B] \frac{\ell}{z, \ell} \quad z \in \mathbb{Z}$$

Let \mathcal{U} be any set including all (finite and infinite) strings over $\mathbb{Z} \cup \{nil, , \}$. Note that $[B]$ is a rule scheme; the ground rules are obtained by instantiating z and ℓ with concrete integers and elements in \mathcal{U} , respectively. We have

$$\widehat{\text{LIST}}(X) = \{z, \ell \mid z \in \mathbb{Z}, \ell \in X\} \cup \{nil\}.$$

The set of possibly infinite (i.e., finite and infinite) lists is the greatest fixed point, $\mathbb{Z}^\infty = \nu \widehat{\text{LIST}}$. We obtain

$$\begin{aligned} \mathbb{Z}^\infty &= \mathcal{U} \cap \widehat{\text{LIST}}(\mathcal{U}) \cap \widehat{\text{LIST}}^2(\mathcal{U}) \cap \dots \\ &= \mathcal{U} \cap (\{z_1, u \mid z_1 \in \mathbb{Z}, u \in \mathcal{U}\} \cup \{nil\}) \\ &\quad \cap (\{z_1, z_2, u \mid z_1, z_2 \in \mathbb{Z}, u \in \mathcal{U}\} \cup \{z_1, nil \mid z_1 \in \mathbb{Z}\} \cup \{nil\}) \\ &\quad \cap \dots \end{aligned}$$

by Kleene's Theorem. The set of infinite lists over \mathbb{Z} is the greatest fixed point of the system consisting only of rule $[B]$, i.e. $\mathbb{Z}^\omega = \nu \widehat{[B]}$. The set of finite lists is the least fixed point $\mu \widehat{\text{LIST}}$. Applying again Kleene's Theorem, we obtain

$$\begin{aligned} \mathbb{Z}^\omega &= \emptyset \cup \widehat{\text{LIST}}(\emptyset) \cup \widehat{\text{LIST}}^2(\emptyset) \cup \dots \\ &= \emptyset \cup \{nil\} \\ &\quad \cup (\{z_1, nil \mid z_1 \in \mathbb{Z}\} \cup \{nil\}) \\ &\quad \cup (\{z_2, z_1, nil \mid z_1, z_2 \in \mathbb{Z}\} \cup \{z_1, nil \mid z_1 \in \mathbb{Z}\} \cup \{nil\}) \\ &\quad \cup \dots \end{aligned}$$

We now give the induction and coinduction principles, which can be used for reasoning about inductively, resp. coinductively defined sets:

Proposition 3.2 (Induction proof principle) *Let \mathcal{U} be a set and $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ be a monotone function. Then*

$$\frac{F(X) \subseteq X}{\mu Y. F(Y) \subseteq X}$$

We shall be using this principle with $F \triangleq \widehat{\mathcal{R}}$ for a set \mathcal{R} of rules. The principle says that if, for a given X , for all rules $(S, x) \in \mathcal{R}$, $S \subseteq X$ implies $x \in X$, then (the set inductively defined by the rules) $\subseteq X$. A set X satisfying $F(X) \subseteq X$ is called *forward closed* w.r.t. F .

Example 3.2 In order to show that the finite list over integers ℓ satisfy a property $p(\ell)$, it is enough to prove that the set $\{\ell \mid p(\ell)\}$ is forward closed, i.e., $\{z, \ell \mid z \in \mathbb{Z} \text{ and } p(\ell)\} \cup \{\text{nil}\} \subseteq \{\ell \mid p(\ell)\}$.

The *coinduction* proof principle is dual to the induction principle:

Proposition 3.3 (Coinduction proof principle) Let \mathcal{U} be a set and $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ be a monotone function. Then

$$\frac{X \subseteq F(X)}{X \subseteq \nu Y. F(Y)}$$

We also use the coinduction proof principle with $F \triangleq \widehat{\mathcal{R}}$ for a set \mathcal{R} of rules. The principle now says that if, for a given X , for all $x \in X$ there is a rule $(S, x) \in \mathcal{R}$ with $S \subseteq X$, then $X \subseteq$ (the set coinductively defined by the rules). A set X satisfying $X \subseteq F(X)$ is called *backward closed* w.r.t. F .

Example 3.3 We consider the lists over integers again. Let $f : \mathcal{U} \rightarrow \mathcal{U}$ be a function. If we want to show that $f(u)$ is a list for each $u \in \mathcal{U}$, it is sufficient to prove that the set $\{f(u) \mid u \in \mathcal{U}\}$ is backward closed, i.e., $\{f(u) \mid u \in \mathcal{U}\} \subseteq \{z, f(u) \mid z \in \mathbb{Z}, u \in \mathcal{U}\} \cup \{\text{nil}\}$.

For the specific case of coinductively defined sets we shall use the fact that membership in such a set amounts to the existence of a certain proof tree:

Definition 3.3 (Proof Trees) Let \mathcal{R} be a set of ground rules over \mathcal{U} and $x \in \mathcal{U}$. A (finite or infinite) tree T is a proof tree of x under \mathcal{R} if it satisfies the following properties:

- the root of T is labelled with x ;
- if y is the label of a node of T and S is the set of labels of the children of this node, then $(S, y) \in \mathcal{R}$.

We often refer the nodes of a proof tree T by their labels. Note that a proof tree can be finite or infinite.

Proposition 3.4 Let \mathcal{R} be a set of a set of ground rules over U such that \mathcal{R} is cocontinuous. Then

$$x \in \mu \widehat{\mathcal{R}} \text{ iff there is a well-founded proof tree of } x \text{ under } \mathcal{R}$$

and

$$x \in \nu \widehat{\mathcal{R}} \text{ iff there is a proof tree of } x \text{ under } \mathcal{R}.$$

Example 3.4 We have $3, 8, \text{nil} \in \nu \widehat{\text{LIST}}$ because there is the following well-founded proof tree under $\widehat{\text{LIST}}$:

$$\begin{array}{c} [A] \frac{\quad}{nil} \\ [B] \frac{[A] \frac{\quad}{nil}}{8, nil} \\ [B] \frac{[B] \frac{[A] \frac{\quad}{nil}}{8, nil}}{3, 8, nil} \end{array}$$

Note that the rule labels are not part of the tree, they are added to emphasise what rule is applied for each node in the tree. We also have that the infinite list $1, 0, 1, 0, 1, 0, \dots$ is in $\nu \widehat{\text{LIST}}$ since there is the following infinite tree under $\widehat{\text{LIST}}$:

$$\begin{array}{c} [B] \frac{\dots}{1, 0, 1, 0, 1, 0, \dots} \\ [B] \frac{[B] \frac{\dots}{1, 0, 1, 0, 1, 0, \dots}}{0, 1, 0, 1, 0, 1, \dots} \\ [B] \frac{[B] \frac{[B] \frac{\dots}{1, 0, 1, 0, 1, 0, \dots}}{0, 1, 0, 1, 0, 1, \dots}}{1, 0, 1, 0, 1, 0, \dots} \end{array}$$

In general, for an element in $\nu \widehat{\mathcal{R}}$ (or $\mu \widehat{\mathcal{R}}$) there may exist more than one proof tree. In [45] games and strategies are used for finding specific proof trees. Here we use the proof trees to define functions on sets (co)inductively defined, i.e., functions of the form $f : A \rightarrow B$, where $A \in \{\mu \widehat{\mathcal{R}}, \nu \widehat{\mathcal{R}}\}$ and $B \in \{\mu \widehat{\mathcal{R}}, \nu \widehat{\mathcal{R}}\}$. According to [31], the functions with $A = \mu \widehat{\mathcal{R}}$ for some \mathcal{R} are called *recursive*, and the functions with $B = \nu \widehat{\mathcal{R}'}$ for some \mathcal{R}' are called *corecursive*. Such functions can be easily defined (specified) using rules (schemes). We assume that \mathcal{R} is defined over \mathcal{U} and \mathcal{R}' over \mathcal{U}' . A *partial proof tree* (under \mathcal{R}') is a tree t with nodes labeled with elements in \mathcal{U}' that can be extended to a proof tree. Let f be a function that sends each rule in \mathcal{R} into a partial proof tree under \mathcal{R}' . This function can be easily extended to proof trees, so we may consider that $f(a) = b$ if the f sends the proof tree of $a \in A$ into the proof tree of $b \in B$. Note that each of these proof trees could be finite (for both inductive and coinductive cases) or infinite (only for the coinductive case).

Here is an example. Let NAT be the set of rules

$$\frac{\quad}{0} \quad \frac{x}{s(x)}$$

Then $\nu \widehat{\text{NAT}}$ is the set of extended natural numbers $\mathbb{N} \cup \{\infty\}$. The function $size : \nu \widehat{\text{LIST}} \rightarrow \nu \widehat{\text{NAT}}$ associating to a list its size can be defined by $size(nil) = 0$, meaning that it maps the rule $\frac{\quad}{nil}$ into the partial tree (that is also a rule in this case) $\frac{\quad}{0}$, and $size(z, \ell) = s(size(\ell))$, meaning that it maps the rule $\frac{\ell}{z, \ell}$ into the partial tree $\frac{size(\ell)}{s(size(\ell))}$. It is interesting that the version for finite lists, $size : \mu \widehat{\text{LIST}} \rightarrow \mu \widehat{\text{NAT}}$, is specified in the same way. Therefore it is important to mention which sets the function is defined on.

3.2 Matching Logic and Reachability Logic

This section is dedicated to presenting the logics that we use in the paper: matching logic and reachability logic. There are several versions of these recently introduced logics, of which we chose those presented in [13].

We start with some notions on algebraic specifications and first-order logic.

A *many-sorted algebraic 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 $T_\Sigma(Var)$ denote the free Σ -algebra of terms with variables, $T_{\Sigma,s}(Var)$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t . For terms t_1, \dots, t_n we let $var(t_1, \dots, t_n)$ denote the set $var(t_1) \cup \dots \cup var(t_n)$. For any *substitution* $\sigma : Var \rightarrow T_\Sigma(Var)$ and term $t \in T_\Sigma(Var)$ 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* $\rho : Var \rightarrow \mathcal{T}$ is extended to a (homonymous) Σ -algebra morphism $\rho : T_\Sigma(Var) \rightarrow \mathcal{T}$. The interpretation of a ground term t in \mathcal{T} is denoted by \mathcal{T}_t . For simplicity, we often write *true*, *false*, $0, 1, \dots$ instead of $\mathcal{T}_{true}, \mathcal{T}_{false}, \mathcal{T}_0, \mathcal{T}_1, \dots$ etc.

Definition 3.4 (Many-Sorted First-Order Logic (FOL)) *Given a set S of sorts, a first-order signature (Σ, Π) 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 (Σ, Π) -model consists of a Σ -algebra \mathcal{T} and a subset $\mathcal{T}_p \subseteq \mathcal{T}_{s_1} \times \dots \times \mathcal{T}_{s_n}$ for each $p \in \Pi_{s_1 \dots s_n}$. Let Var denote a S -sorted set of variables. The set of (Σ, Π) -formulas is defined by*

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

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

- (1) $\rho \models \top$;
- (2) $\rho \models p(t_1, \dots, t_n)$ iff $(t_1\rho, \dots, t_n\rho) \in \mathcal{T}_p$;
- (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.

Matching Logic

We next recall the syntax and semantics of matching logic (ML) [38] and reachability logic (RL) [13]. ML is a static logic of configurations, whereas RL is a dynamic logic of configurations, expressing their evolution over time. Here we consider only the topmost version of ML. 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 3.5 (ML signature) *An ML signature is a triple $\Phi = (\Sigma, \Pi, Cfg)$, where (Σ, Π) is a many-sorted first-order signature and Cfg is a distinguished sort in Σ for configurations. The set of ML formulas over Φ is defined by*

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

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

Definition 3.6 (Basic, Elementary, Quantified Elementary Patterns)

Let $\Phi = (\Sigma, \Pi, Cfg)$ be an ML signature. A basic pattern is a term $\pi \in T_{\Sigma, Cfg}(Var)$. An elementary pattern is an 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 quantified elementary pattern is an ML formula of the form $(\exists X)\pi \wedge \phi$ with $X \subset Var$ and $\pi \wedge \phi$ an elementary pattern. We identify basic patterns π with corresponding elementary patterns $\pi \wedge true$ and with quantified elementary patterns $(\exists \emptyset)\pi \wedge \phi$.

A basic pattern π thus defines a set of (concrete) configurations, and the condition ϕ gives additional constraints these configurations must satisfy. If present, existentially quantified variables are used to abstract away some details from the pattern. Examples of CinK patterns are the basic pattern

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

and the elementary pattern

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

An example of a CinK quantified elementary pattern is given below; it will occur in the sequel and its meaning will be explained at the appropriate place:

$$(\exists l, g) \langle\langle \cdot \rangle_k \langle (x \mapsto l)E \rangle_{env} \langle (l \mapsto g)S \rangle_{store} \rangle_{cfg} \wedge g = gcd(a, b)$$

Definition 3.7 (ML Model) *A model for ML signature $\Phi = (\Sigma, \Pi, Cfg)$ is a (Σ, Π) first-order model \mathcal{T} . Concrete configurations (or simply configurations) are elements of \mathcal{T}_{Cfg} , i.e., \mathcal{T} -interpretations of ground terms of sort Cfg .*

Hereafter we fix an ML signature $\Phi = (\Sigma, \Pi, Cfg)$ and a model \mathcal{T} for it.

Definition 3.8 (ML Satisfaction) *The satisfaction relation \models relates pairs (γ, ρ) , where $\gamma \in \mathcal{T}_{\text{Cfg}}$ and $\rho : \text{Var} \rightarrow \mathcal{T}$, with Φ -formulas φ . For basic patterns π , $(\gamma, \rho) \models \pi$ is defined by $\gamma = \pi\rho$. For the remaining ML constructions satisfaction is defined as expected, e.g., $(\gamma, \rho) \models \exists X\varphi$ iff $(\gamma, \rho') \models \varphi$ for some $\rho' : \text{Var} \rightarrow \mathcal{T}$ such that $x\rho = x\rho'$ for all $x \in \text{Var} \setminus X$. If φ is an ML formula then $\llbracket \varphi \rrbracket$ denotes the set of concrete configurations $\{\gamma \mid (\gamma, \rho) \models \varphi \text{ for some } \rho\}$.*

The following encoding of ML into FOL will be used in the rest of the paper.

Definition 3.9 (FOL encoding of ML [13]) *If φ is an ML formula then $\varphi^{=?}$ is the FOL formula $(\exists z)\varphi'$, where φ' is obtained from φ by replacing each basic pattern occurrence π with $z = \pi$, and z is a variable that does not occur in φ .*

Example 3.5 *Here are a few examples of ML formulas and their FOL encodings:*

φ	$\varphi^{=?}$
$(\pi_1 \wedge \phi_1) \vee (\pi_2 \wedge \phi_2)$	$(\exists z)((z = \pi_1 \wedge \phi_1) \vee (z = \pi_2 \wedge \phi_2))$
$\neg\pi$	$(\exists z)\neg(z = \pi)$
$\pi_1 \wedge \neg\pi_2$	$(\exists z)((z = \pi_1) \wedge \neg(z = \pi_2))$
$\pi \vee \neg\pi$	$(\exists z)(z = \pi \vee \neg(z = \pi))$

The relationship between ML formulas and their FOL encodings is now given.

Proposition 3.5 ([13]) *Let φ be an ML formula and $\rho : \text{Var} \rightarrow \mathcal{T}$. Then $\rho \models \varphi^{=?}$ iff there is γ such that $(\gamma, \rho) \models \varphi$.*

Reachability Logic

Definition 3.10 *An RL formula is a pair of ML formulas $\varphi_1 \Rightarrow \varphi_2$.*

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

Example 3.6 Consider the program `gcd` from Figure 4. The following RL formula

$$\begin{aligned} & \langle \langle \text{gcd} \rangle_k \langle \mathbf{a} \mapsto l_1 \ \mathbf{b} \mapsto l_2 \ \dots \rangle_{\text{env}} \langle l_1 \mapsto a \ l_2 \mapsto b \ \dots \rangle_{\text{store}} \ \dots \rangle_{\text{cfg}} \wedge a \geq 0 \wedge b \geq 0 \Rightarrow \\ & (\exists l, g) \langle \langle \cdot \rangle_k \langle \mathbf{x} \mapsto l \ \dots \rangle_{\text{env}} \langle l \mapsto g \ \dots \rangle_{\text{store}} \ \dots \rangle_{\text{cfg}} \wedge g = \text{gcd}(a, b) \end{aligned} \quad (2)$$

specifies that finite complete executions of the program `gcd` from configurations where the program variables `a`, `b` are bound to non-negative values a , b reach configurations where `x` is bound to a location l where an integer value g such that $g = \text{gcd}(a, b)$ is stored. Here, $\text{gcd}()$ is a mathematical definition of the greatest-common-divisor operation (with $\text{gcd}(0, 0) = 0$ by convention).

Also by convention the indices from data-domain predicates such as \geq_{Int} are dropped to simplify the presentation, and variables occurring in both left and right-hand sides of a formula, but are irrelevant to the formula's meaning, are replaced by ellipses (e.g., the rest of the store and environment cells).

RL formulas are interpreted over transition systems generated by *language definitions*. Next, we introduce that concept and give the semantics of RL.

3.3 Language Definitions

Our symbolic-execution approach is independent of the formal framework used for defining languages as well as from the languages being defined. We thus use 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 an ML signature (Σ, Π, Cfg) giving syntax to the language's execution infrastructure (called *configuration*). Cfg is the sort for configurations.
- A Φ -Model \mathcal{T} . Recall that \mathcal{T}_s denote the elements of the model \mathcal{T} that have the sort s , in particular, the elements of \mathcal{T}_{Cfg} are called *configurations*.
- A set \mathcal{S} of RL formulas, defining the operational semantics of the language.

In the rest of the paper we shall also be using a fragment of the CinK language (shown in Section 2), called `WHILE`, whose definition is simpler and therefore is easier to use for small yet illustrative examples. The syntax of `WHILE` consists only of expressions, assignments, sequences and blocks of statements, and while loops. For simplicity, the configuration for `WHILE` consists of only two cells $\langle \rangle_k$ and $\langle \rangle_{env}$ which contain the program to be executed, and respectively, a map from program variables to their values: $\langle \langle Pgm \rangle_k \langle Map \rangle_{env} \rangle_{cfg}$. The semantics of expressions in `WHILE` is given using the same rules as in CinK. However, we choose a slightly different semantics for some statements, namely for assignment and lookup (due to the new configuration structure), and for the `while` loop:

$$\begin{aligned}
\langle X \ \dots \rangle_k \langle \dots X \mapsto I \ \dots \rangle_{env} &\Rightarrow \langle I \ \dots \rangle_k \langle \dots X \mapsto I \ \dots \rangle_{env} && [\text{WHILE_lookup}] \\
\langle X = I ; \ \dots \rangle_k \langle \dots X \mapsto _ \ \dots \rangle_{env} &\Rightarrow \langle \dots \rangle_k \langle \dots X \mapsto I \ \dots \rangle_{env} && [\text{WHILE_assign}] \\
\langle \text{while}(E) S \ \dots \rangle_k \langle \sigma \rangle_{env} \wedge \sigma[E] =_{Bool} \text{false} &\Rightarrow \langle \dots \rangle_k \langle \sigma \rangle_{env} && [\text{WHILE_true}] \\
\langle \text{while}(E) S \ \dots \rangle_k \langle \sigma \rangle_{env} \wedge \sigma[E] =_{Bool} \text{true} &\Rightarrow \\
&\langle S \ \text{while}(E) S \ \dots \rangle_k \langle \sigma \rangle_{env} && [\text{WHILE_false}]
\end{aligned}$$

The while loop statement has two corresponding rules in the semantics, one for each possible evaluation of the condition E : if the evaluation of the condition

E in state σ , denoted $\sigma[E]$, is *true*, then execute once the body of the loop and the loop again; otherwise, continue executing the rest of the program. The definition of `while` exhibits the flexibility of the \mathbb{K} Framework to define the semantics of a language in various styles.

We now show how `while` fits into our notion of language definition. The ML signature (Σ, Π, Cfg) corresponding to `while` contains all the sorts and operations defined in its syntax. Nonterminals in the syntax are sorts and syntax productions are operations. For instance, Σ includes a sort for expressions Exp and its corresponding operations, e.g., $_+ _ : Exp \times Exp \rightarrow Exp$, $_ / _ : Exp \times Exp \rightarrow Exp$, etc. The ML signature also contains a sort Cfg for configurations, which has a unique constructor $Stmt \times Map \rightarrow Cfg$ denoted by $\langle \langle _ \rangle_k \langle _ \rangle_{env} \rangle_{cfg}$ (the occurrences of the symbol $_$ show the places of the arguments). The model \mathcal{T} interprets all the constants and operations from the signature. By \mathcal{T}_1 and \mathcal{T}_{true} we denote the interpretation of the constant symbols `1` and *true*. The set \mathcal{S} of RL formulas contains the rules shown above (for assignment, lookup, and `while`), the rules corresponding to expressions from Figure 3, and rules generated by \mathbb{K} from strictness annotations.

Language definitions generate *execution paths* obtained by applying rules in \mathcal{S} to configurations in \mathcal{T}_{Cfg} . We give the relevant definitions using coinduction.

Notation We write $\gamma_1 \Rightarrow_{\mathcal{S}}^{\rho} \gamma_2$ iff there exists $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ and a renaming σ of the variables in $\varphi_1 \Rightarrow \varphi_2$ such that $(\gamma_i, \sigma\rho) \models \varphi_i$, $i = 1, 2$, and $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ iff there is ρ such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\rho} \gamma_2$.

Definition 3.11 (Execution Paths) *Let $(\Phi, \mathcal{T}, \mathcal{S})$ be a language definition. The set of execution paths (τ, ρ) is coinductively defined by the following set of rules:*

$$\frac{}{(\gamma, \rho)} \qquad \frac{(\tau, \rho)}{(\gamma_0 \Rightarrow_{\mathcal{S}} \tau, \rho)} \gamma_0 \Rightarrow_{\mathcal{S}}^{\rho} hd(\tau)$$

where $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \rightarrow \mathcal{T}$, and hd is coinductively defined by

$$hd(\gamma) = \gamma \qquad hd(\gamma_0 \Rightarrow_{\mathcal{S}} \tau) = \gamma_0$$

We say that (τ, ρ) starts from φ if $(hd(\tau), \rho) \models \varphi$.

Note that this set of rules is akin to the one for lists (shown in Example 3.1); an execution path can be seen as a list of concrete configurations, where every two consecutive elements γ and γ' satisfy $\gamma \Rightarrow_{\mathcal{S}}^{\rho} \gamma'$. The largest set satisfied by these rules is the set of all finite and infinite execution paths.

Example 3.7 *Let $\rho : Var \rightarrow \mathcal{T}$ be a valuation such that $X\rho = \mathbf{x}$ and $I\rho = 2$, where X and I are variables from the assignment rule for `while`. Then*

$$\langle \langle \mathbf{x} = 2; \rangle_k \langle \mathbf{x} \mapsto 3 \rangle_{env} \rangle_{cfg} \Rightarrow_{\mathcal{S}} \langle \langle \cdot \rangle_k \langle \mathbf{x} \mapsto 2 \rangle_{env} \rangle_{cfg}$$

is a (finite) execution path obtained using the rules from Definition 3.11. Execution paths can be infinite too, e.g.,

$$\begin{aligned}
\tau &\triangleq \langle\langle \mathbf{while}(true) \mathbf{x}=1; \rangle_k \langle \mathbf{x} \mapsto 0 \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow_{\mathcal{S}} \\
&\quad \langle\langle \mathbf{x}=1; \mathbf{while}(true) \mathbf{x}=1; \rangle_k \langle \mathbf{x} \mapsto 0 \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow_{\mathcal{S}} \\
&\quad \langle\langle \mathbf{while}(true) \mathbf{x}=1; \rangle_k \langle \mathbf{x} \mapsto 1 \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow_{\mathcal{S}} \\
&\quad \langle\langle \mathbf{x}=1; \mathbf{while}(true) \mathbf{x}=1; \rangle_k \langle \mathbf{x} \mapsto 1 \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow_{\mathcal{S}} \\
&\quad \langle\langle \mathbf{while}(true) \mathbf{x}=1; \rangle_k \langle \mathbf{x} \mapsto 1 \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow_{\mathcal{S}} \\
&\quad \dots
\end{aligned}$$

which is obtained by applying the rules for **while** and assignment repeatedly.

Remark 3.1 The above definition for execution paths is not quite standard since it includes a fixed valuation ρ . A more natural definition is

$$\frac{}{\gamma} \qquad \frac{\tau}{\gamma_0 \Rightarrow_{\mathcal{S}} \tau} \quad \gamma_0 \Rightarrow_{\mathcal{S}} hd(\tau)$$

Recall that the variables occurring in rules can be renamed when they are used in execution steps (see the notation preceding Definition 3.11) and hence for each execution path $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \dots$, defined by the above rules, there is a valuation ρ and the execution path $(\gamma_0, \rho) \Rightarrow_{\mathcal{S}} (\gamma_1, \rho) \Rightarrow_{\mathcal{S}} \dots$ defined by Definition 3.11. We preferred the version with a fixed valuation since it simplifies many proofs.

We shall also be needing the following two notions. A configuration is *irreducible* if there is no transition outgoing from it:

Definition 3.12 (Irreducible Configurations) Let $(\Phi, \mathcal{T}, \mathcal{S})$ be a language definition. We say that $\gamma \in \mathcal{T}_{\text{cfg}}$ is irreducible iff there is no γ' such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$.

An execution path is *complete* if either it is finite and ends up in an irreducible configuration, or it is infinite.

Definition 3.13 (Complete Execution Paths) Let $(\Phi, \mathcal{T}, \mathcal{S})$ be a language definition. The set of complete execution paths is coinductively defined by the following set of rules:

$$\frac{}{(\gamma, \rho) \text{ irreducible}} \qquad \frac{(\tau, \rho)}{(\gamma_0 \Rightarrow_{\mathcal{S}} \tau, \rho)} \quad \gamma_0 \Rightarrow_{\mathcal{S}}^{\rho} hd(\tau)$$

Example 3.8 Recall $\gamma' \triangleq \langle\langle \cdot \rangle_k \langle \mathbf{x} \mapsto 2 \rangle_{\text{env}} \rangle_{\text{cfg}}$ from the previous example. Then, γ' is irreducible because there is no rule in the semantics of **WHILE** that can be applied to it. Another example of irreducible configuration is

$$\gamma'' \triangleq \langle\langle 10/0 \rangle_k \langle \mathbf{x} \mapsto 2 \ \mathbf{y} \mapsto 1 \rangle_{\text{env}} \rangle_{\text{cfg}}.$$

Since the side condition of division rule does not hold in this case, the rule cannot be applied and thus, there are no successors for γ'' .

From γ' irreducible we obtain that any path $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ is complete. Note that infinite execution paths are complete as well; for instance, the infinite path τ from Example 3.7 is complete.

Next, we define the notion of *derivative* of ML and RL formulas needed to define semantics of RL and symbolic execution (in Section 4). We shall see in Section 4 that this rather technical definition encodes the set of all concrete successors of configurations that satisfy a given ML formula. In Section 6 we shall also see that it amounts (under reasonable conditions) to standard rewriting with a modified set of rules.

Definition 3.14 (Derivatives for ML and RL Formulas) *If φ is an ML formula then $\Delta_{\mathcal{S}}(\varphi) \triangleq \{(\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi)^{=?} \wedge \varphi_r \mid \varphi_l \Rightarrow \varphi_r \in \mathcal{S}\}$. If $\varphi \Rightarrow \varphi'$ is an RL formula then $\Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi') \triangleq \{\varphi_1 \Rightarrow \varphi' \mid \varphi_1 \in \Delta_{\mathcal{S}}(\varphi)\}$.*

Example 3.9 *Let $\varphi \triangleq \langle\langle X=I'; \curvearrowright C' \rangle_k \langle X' \mapsto V' M' \rangle_{\text{env}} \rangle_{\text{cfg}}$ be an ML formula and $\varphi_l \Rightarrow \varphi_r$ be the RL formula corresponding to assignment in WHILE:*

$$\langle\langle X=I; \curvearrowright C \rangle_k \langle X \mapsto V M \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow \langle\langle C \rangle_k \langle X \mapsto I M \rangle_{\text{env}} \rangle_{\text{cfg}}$$

The derivative $\varphi' \triangleq \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi)$ specifies the configurations obtained from those satisfying the initial formula φ after executing the assignment rule:

$$\begin{aligned} \varphi' &\triangleq (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi)^{=?} \wedge \varphi_r \\ &= (\exists X, I, C, V, M) (\langle\langle X=I; \curvearrowright C \rangle_k \langle X \mapsto V M \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge \\ &\quad \langle\langle X'=I'; \curvearrowright C' \rangle_k \langle X' \mapsto V' M' \rangle_{\text{env}} \rangle_{\text{cfg}})^{=?} \wedge \langle\langle C \rangle_k \langle X \mapsto I M \rangle_{\text{env}} \rangle_{\text{cfg}} \\ &= (\exists X, I, C, V, M) (\exists z) (z = \langle\langle X=I; \curvearrowright C \rangle_k \langle X \mapsto V M \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge \\ &\quad z = \langle\langle X'=I'; \curvearrowright C' \rangle_k \langle X' \mapsto V' M' \rangle_{\text{env}} \rangle_{\text{cfg}}) \wedge \langle\langle C \rangle_k \langle X \mapsto I M \rangle_{\text{env}} \rangle_{\text{cfg}} \\ &= (\exists X, I, C, V, M) (\langle\langle X=I; \curvearrowright C \rangle_k \langle X \mapsto V M \rangle_{\text{env}} \rangle_{\text{cfg}} = \\ &\quad \langle\langle X'=I'; \curvearrowright C' \rangle_k \langle X' \mapsto V' M' \rangle_{\text{env}} \rangle_{\text{cfg}}) \wedge \langle\langle C \rangle_k \langle X \mapsto I M \rangle_{\text{env}} \rangle_{\text{cfg}} \end{aligned}$$

which simplifies to

$$\varphi' = \langle\langle C \rangle_k \langle X' \mapsto I' M' \rangle_{\text{env}} \rangle_{\text{cfg}}$$

using the fact that the equality

$$\langle\langle X=I; \curvearrowright C \rangle_k \langle X \mapsto V M \rangle_{\text{env}} \rangle_{\text{cfg}} = \langle\langle X'=I'; \curvearrowright C' \rangle_k \langle X' \mapsto V' M' \rangle_{\text{env}} \rangle_{\text{cfg}}$$

implies $X = X'$, $I = I'$, $C = C'$, $M = M'$, and $V = V'$. That is, after performing the assignment of $X' = I'$, X' is bound to I' in the environment.

In the paper we use the notions of \mathcal{S} -derivability of an ML formula φ (also extended to RL formulas $\varphi \Rightarrow \varphi'$), and *totality* of a set \mathcal{S} of RL formulas:

Definition 3.15 (\mathcal{S} -derivability) *An ML formula φ is \mathcal{S} -derivable if $\Delta_{\mathcal{S}}(\varphi)$ is satisfiable. An RL formula $\varphi \Rightarrow \varphi'$ is \mathcal{S} -derivable if φ is \mathcal{S} -derivable.*

Example 3.10 Recall φ, φ' and $\varphi_l \Rightarrow \varphi_r$ from Example 3.9. The ML formula φ is \mathcal{S} -derivable since $\varphi' \triangleq \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi) (\in \Delta_{\mathcal{S}}(\varphi))$ is satisfiable (we have $\gamma' \in \llbracket \varphi' \rrbracket$, where $\gamma' \triangleq \langle \langle \cdot \rangle_{\mathbf{k}} \langle \mathbf{x} \mapsto 2 \rangle_{\text{env}} \rangle_{\text{cfg}}$).

On the other hand, the ML formula $\widehat{\varphi} = \langle \langle I'_1 / I'_2; \curvearrowright C' \rangle_{\mathbf{k}} \langle E' \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge I'_2 =_{\text{Int}} 0$ is not \mathcal{S} -derivable. In fact, for any rule $\varphi_l \Rightarrow \varphi_r \in \mathcal{S}$, the derivative $\Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\widehat{\varphi})$ is not satisfiable. For example, let $\varphi_l \Rightarrow \varphi_r$ be the division rule:

$$\langle \langle I_1 / I_2 \curvearrowright C \rangle_{\mathbf{k}} \langle E \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge I_2 \neq_{\text{Int}} 0 \Rightarrow \langle \langle I_1 / \text{Int} I_2 \curvearrowright C \rangle_{\mathbf{k}} \langle E \rangle_{\text{env}} \rangle_{\text{cfg}}$$

The derivative $\Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\widehat{\varphi})$ is:

$$\begin{aligned} (\exists I_1, I_2, C, E) & \left(\langle \langle I_1 / I_2 \curvearrowright C \rangle_{\mathbf{k}} \langle E \rangle_{\text{env}} \rangle_{\text{cfg}} = \langle \langle I'_1 / I'_2; \curvearrowright C' \rangle_{\mathbf{k}} \langle E' \rangle_{\text{env}} \rangle_{\text{cfg}} \right. \\ & \wedge I_2 \neq_{\text{Int}} 0 \wedge I'_2 =_{\text{Int}} 0 \\ & \left. \right) \wedge \langle \langle I_1 / \text{Int} I_2 \curvearrowright C \rangle_{\mathbf{k}} \langle E \rangle_{\text{env}} \rangle_{\text{cfg}} \end{aligned}$$

$\Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\widehat{\varphi})$ is not satisfiable because there is no I_2 which simultaneously satisfies $I_2 = I'_2$ (implied by $\langle \langle I_1 / I_2 \curvearrowright C \rangle_{\mathbf{k}} \langle E \rangle_{\text{env}} \rangle_{\text{cfg}} = \langle \langle I'_1 / I'_2; \curvearrowright C' \rangle_{\mathbf{k}} \langle E' \rangle_{\text{env}} \rangle_{\text{cfg}}$), $I_2 \neq_{\text{Int}} 0$, and $I'_2 =_{\text{Int}} 0$.

Definition 3.16 (Totality) A set \mathcal{S} of RL formulas is total iff for each \mathcal{S} -derivable φ and each (γ, ρ) such that $(\gamma, \rho) \models \varphi$, there is γ_1 such that $\gamma \Rightarrow_{\mathcal{S}} \gamma_1$.

Note the difference between \mathcal{S} -derivability and totality: \mathcal{S} -derivability requires to have at least one transition starting from φ and the totality requires to have at least one transition starting from γ for any model (γ, ρ) of φ .

Remark 3.2 The semantics of CinK is not total because of the rules for division and modulo. The rule for division: $\langle \langle I_1 / I_2 \curvearrowright C \rangle_{\mathbf{k}} \rangle_{\text{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle \langle I_1 / \text{Int} I_2 \curvearrowright C \rangle_{\mathbf{k}} \rangle_{\text{cfg}}$ does not meet the condition of Definition 3.16 because of the condition $I_2 \neq 0$ which restricts the application of the rule. The semantics can be made total by adding a rule $\langle \langle I_1 / I_2 \curvearrowright C \rangle_{\mathbf{k}} \rangle_{\text{cfg}} \wedge I_2 = 0 \Rightarrow \langle \langle \text{error} \curvearrowright C \rangle_{\mathbf{k}} \rangle_{\text{cfg}}$ that leads divisions by zero into “error” configurations. In this way, for any γ which is an instance of $\langle \langle I_1 / I_2 \curvearrowright C \rangle_{\mathbf{k}} \rangle_{\text{cfg}}$ there is at least one rule in the semantics which can be applied.

We assume hereafter that the CinK semantics (and implicitly that of WHILE) has been transformed as above.

We can now define the semantics of RL, in a different, yet equivalent manner to the original definition given in [13]. Again, we are using coinduction:

Definition 3.17 (Semantics of RL) Let (τ, ρ) be an execution path and $\varphi \Rightarrow \varphi'$ an RL formula. We say that (τ, ρ) satisfies $\varphi \Rightarrow \varphi'$ and we write $(\tau, \rho) \models \varphi \Rightarrow \varphi'$ if $\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle \in \nu(3, 4)$, where $(3, 4)$ is the system of

ground rules:

$$\frac{}{\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle} (hd(\tau), \rho) \models \varphi \wedge \varphi' \quad (3)$$

$$\frac{\langle (\tau, \rho), \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi \Rightarrow \varphi') \rangle}{\langle (\gamma_0 \Rightarrow_S \tau, \rho), \varphi \Rightarrow \varphi' \rangle} (\gamma_0, \rho) \models (\varphi_l \wedge \varphi), \varphi_l \Rightarrow \varphi_r \in \mathcal{S} \quad (4)$$

$\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff $(\tau, \rho) \models \varphi \Rightarrow \varphi'$ for each complete (τ, ρ) starting from φ . If F is a set of RL formulas then $\mathcal{S} \models F$ iff $\mathcal{S} \models \varphi \Rightarrow \varphi'$ for each $\varphi \Rightarrow \varphi' \in F$. We let $\llbracket \varphi \Rightarrow \varphi' \rrbracket \triangleq \{ \tau \mid (\exists \rho)(\tau, \rho) \models \varphi \Rightarrow \varphi' \}$ and $\llbracket F \rrbracket = \bigcup_{\varphi \Rightarrow \varphi' \in F} \llbracket \varphi \Rightarrow \varphi' \rrbracket$.

Example 3.11 According to Example 3.9 and Example 3.10 we obtain

$$\begin{aligned} & \langle \langle \mathbf{x} = 2; \rangle_k \langle \mathbf{x} \mapsto 3 \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow_S \langle \langle \cdot \rangle_k \langle \mathbf{x} \mapsto 2 \rangle_{\text{env}} \rangle_{\text{cfg}}, \rho \\ & \models \\ & \langle \langle X' = I'; \curvearrowright C' \rangle_k \langle X' \mapsto V' M' \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow \langle \langle C' \rangle_k \langle X' \mapsto I' M' \rangle_{\text{env}} \rangle_{\text{cfg}} \end{aligned}$$

with $\rho : \text{Var} \rightarrow \mathcal{T}$ a valuation such that $X'\rho = \mathbf{x}$, $I'\rho = 2$, $C'\rho = \cdot$, and $V'\rho = 3$.

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.

We consider given a language definition $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$, with $\Phi = (\Sigma, \Pi)$ and Cfg the sort for configurations. Since our goal is to relate concrete and symbolic executions we need to define first what *symbolic execution paths* are.

Definition 4.1 (Symbolic Execution Path) *The set of symbolic execution paths is coinductively defined by the following set of rules:*

$$\frac{}{\varphi} \varphi \text{ satisfiable} \quad (5)$$

$$\frac{\tau^s}{\varphi_0 \Rightarrow_S^s \tau^s} \varphi_0 \text{ } \mathcal{S}\text{-derivable, } hd(\tau^s) \in \Delta_{\mathcal{S}}(\varphi_0) \quad (6)$$

where hd is coinductively defined by

$$hd(\varphi) = \varphi \qquad hd(\varphi_0 \Rightarrow_{\mathcal{S}}^5 \tau^5) = \varphi_0$$

Remark 4.1 An alternative, equivalent definition for symbolic paths consists in requiring that $hd(\tau^5)$ is satisfiable in the rule (6) above. As a consequence, the symbolic paths thus defined are all feasible.

A symbolic execution path essentially consists in a possibly infinite list of ML formulas, where every two consecutive formulas φ and φ' satisfy $\varphi' \in \Delta_{\mathcal{S}}(\varphi)$.

Notation We write $\varphi_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}}^5 \varphi_2$ if $\varphi_2 = \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi_1)$.

A symbolic execution path *covers* a concrete one if the latter is an instance of the former. Formally, this notion is defined below:

Definition 4.2 Consider the following rules:

$$\frac{}{\langle \varphi, (\gamma, \rho) \rangle} (\gamma, \rho) \models \varphi \qquad (7)$$

$$\frac{\langle \tau^5, (\tau, \rho) \rangle}{\langle \varphi_0 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}}^5 \tau^5, (\gamma_0 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \tau, \rho) \rangle} (\gamma_0, \rho) \models \varphi_0 \qquad (8)$$

We write $\tau^5 \sqsupseteq (\tau, \rho)$ if $\langle \tau^5, (\tau, \rho) \rangle \in \nu(\widehat{7, 8})$, where (7, 8) is the system of the ground rules defined by (7) and (8).

Note that if $\tau^5 \sqsupseteq (\tau, \rho)$ then the symbolic path and the concrete one either are finite and have the same length or both are infinite.

We now show that the symbolic execution thus defined is related with concrete execution via the coverage and precision properties stated in the introduction. The coverage property states that the symbolic paths symbolically simulate concrete ones.

Lemma 4.1 (Symbolic Step Forward-Simulates Concrete Step)

If $\gamma_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma_2$ and $(\gamma_1, \rho) \models \varphi_1$ then there is φ_2 such that $(\gamma_2, \rho) \models \varphi_2$ and $\varphi_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}}^5 \varphi_2$.

The next theorem 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.

Theorem 4.1 (Coverage) For every concrete path $\tau \triangleq (\gamma \Rightarrow_{\mathcal{S}} \tau, \rho)$ and ML formula φ such that $(\gamma, \rho) \models \varphi$ there exists a symbolic path $\tau^5 \triangleq (\varphi \Rightarrow_{\mathcal{S}}^5 \tau^5)$ such that $\tau^5 \sqsupseteq (\tau, \rho)$.

The precision property states that symbolic execution paths are backwards-simulated by concrete ones. Forward simulation does not hold in this case, because the formulas resulting from a symbolic step may be unsatisfiable.

Lemma 4.2 (Concrete Step Backward-Simulates Symbolic Step)

If $\varphi_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}}^s \varphi_2$ and $(\gamma_2, \rho) \models \varphi_2$ then there is γ_1 such that $\gamma_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma_2$ and $(\gamma_1, \rho) \models \varphi_1$.

The precision result, given below, is based on above the backwards-simulation lemma. Path executed "backwards" are finite because they end up in an initial state, thus, the precision result holds for finite paths only.

Theorem 4.2 (Precision) For all finite symbolic executions τ^s , there exists (τ, ρ) such that $\tau^s \sqsubseteq (\tau, \rho)$.

The theorems in this section generalise the similar ones in [4] and say that symbolic execution can be used as a sound program-analysis technique. That is, to check whether a given (finite) control-flow path is concretely executable in a program, one can attempt to symbolically execute the rules corresponding to the instructions in the control-flow path; thanks to the coverage and precision results, the attempt succeeds if and only if the path is concretely executable.

5 Application: Reachability-Logic Verification

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 (2) on the program in Fig. 4 require such an unbounded-length symbolic executions because of the unboundedly many iterations of the loop. This issue is dealt with in the present section by means of *circular coinduction* built on top of symbolic execution.

We first define the notion of symbolic semantics of RL formulas, then rephrase RL formula verification by symbolic execution as a proof system. Finally, circular coinduction amounts to adding a *circularity* rule to that proof system. We prove that the resulting 3-rule proof system is sound, i.e., that if it manages to prove a given (set of) RL formula(s) then the formulas hold semantically.

Definition 5.1 (Symbolic Semantics of RL) Let τ^s be a symbolic execution path and $\varphi \Rightarrow \varphi'$ and RL formula. We say that τ^s satisfies $\varphi \Rightarrow \varphi'$ and we write $\tau^s \models \varphi \Rightarrow \varphi'$ for $\langle \tau^s, \varphi \Rightarrow \varphi' \rangle \in \nu(9, 10)$, where $(9, 10)$ is the system

of the ground rules:

$$\frac{}{\langle \varphi_0, \varphi \Rightarrow \varphi' \rangle} \mathcal{T} \models \varphi_0 \rightarrow \varphi \wedge \varphi' \quad (9)$$

$$\frac{\langle \tau^s, \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi') \rangle}{\langle \varphi_0 \Rightarrow_{\varphi_1 \Rightarrow \varphi_2}^s \tau^s, \varphi \Rightarrow \varphi' \rangle} \mathcal{T} \models \varphi_0 \rightarrow \varphi, \varphi_1 \Rightarrow \varphi_2 \in \mathcal{S} \quad (10)$$

The following proof system can be used to prove RL formulas by symbolic execution. The first rule says that RL formulas hold if their left hand-side implies (in the sense of ML) their right-hand side. The second rule says that (derivable) formulas hold whenever their derivatives hold.

Definition 5.2 (SYSTEP)

$$[\text{impl}] \frac{}{\varphi \Rightarrow \varphi'} \mathcal{T} \models \varphi \rightarrow \varphi' \quad [\text{der}] \frac{\Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi')}{\varphi \Rightarrow \varphi'} \varphi \text{ is } \mathcal{S}\text{-derivable}$$

The following theorem states the soundness of this simple proof system:

Theorem 5.1 *If \mathcal{S} is total then $\mathcal{S} \models \nu \widehat{\text{SYSTEP}}$.*

Specifically, if one can construct a finite proof tree under SYSTEP for a given RL formula then the formula belongs to $\nu \widehat{\text{SYSTEP}}$. Theorem 5.1 then says that the formula holds in the semantics \mathcal{S} . An RL formula also holds if an infinite proof tree under SYSTEP can be built. We give below an example of an infinite proof tree, which we reuse in order to show how *circular coinduction* can "fold" infinite proof trees into finite ones in a stronger proof system.

Suppose that \mathcal{S} is the set of RL formulas corresponding WHILE language. A proof tree for

$$\begin{aligned} & \langle \langle \text{while } (x \neq 0) \{s = s + x; x = x - 1;\} \curvearrowright P \rangle_k \langle x \mapsto a \text{ } s \mapsto 0 \rangle_{\text{env}} \rangle_{\text{cfg}} \Rightarrow \\ & (\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \text{ } s \mapsto b \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge b =_{\text{Int}} \frac{a(a + \text{Int } 1)}{2} \end{aligned}$$

is represented in Figure 5. T_1 corresponds to the case when the `while` condition is false. One can see that T_2 is infinite and corresponds to unfolding indefinitely of the `while` loop.

5.1 Circular Coinduction

In this section we show how to reduce infinite proof trees to finite ones in a stronger proof system, which adds to SYSTEP a *circularity* rule. The rule is thus called because it allows one to use *conclusions*, i.e., formulas to be proved (from a set G of *goals*) as *hypotheses* during proofs of formulas from the set G .

T_2

$$\begin{array}{c}
\text{[der]} \frac{\dots}{\left\langle \begin{array}{l} \langle \text{while } (x \neq 0) \{s=s+x; x=x-1;\} P \rangle_k \\ \langle x \mapsto a -_{Int} 1 \ s \mapsto a \rangle_{env} \end{array} \right\rangle_{cfg} \wedge a \neq_{Int} 0} \\
\Rightarrow \\
(\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{env} \rangle_{cfg} \wedge b =_{Int} \frac{a(a +_{Int} 1)}{2} \\
\text{[der]} \frac{}{\left\langle \begin{array}{l} \langle x=x-1; \text{while } (x \neq 0) \{s=s+x; x=x-1;\} P \rangle_k \\ \langle x \mapsto a \ s \mapsto a \rangle_{env} \end{array} \right\rangle_{cfg} \wedge a \neq_{Int} 0} \\
\Rightarrow \\
(\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{env} \rangle_{cfg} \wedge b =_{Int} \frac{a(a +_{Int} 1)}{2} \\
\text{[der]} \frac{}{\left\langle \begin{array}{l} \langle s=s+x; x=x-1; \text{while } (x \neq 0) \{s=s+x; x=x-1;\} P \rangle_k \\ \langle x \mapsto a \ s \mapsto 0 \rangle_{env} \end{array} \right\rangle_{cfg} \wedge a \neq_{Int} 0} \\
\Rightarrow \\
(\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{env} \rangle_{cfg} \wedge b =_{Int} \frac{a(a +_{Int} 1)}{2}
\end{array}$$

T_1

$$\text{[impl]} \frac{}{\langle \langle P \rangle_k \langle x \mapsto a \ s \mapsto 0 \rangle_{env} \rangle_{cfg} \wedge a =_{Int} 0 \Rightarrow (\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{env} \rangle_{cfg} \wedge b =_{Int} \frac{a(a +_{Int} 1)}{2}}$$

$$\frac{T_1 \quad T_2}{\langle \langle \text{while } (x \neq 0) \{s=s+x; x=x-1;\} \curvearrowright P \rangle_k \langle x \mapsto a \ s \mapsto 0 \rangle_{env} \rangle_{cfg} \Rightarrow (\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{env} \rangle_{cfg} \wedge b =_{Int} \frac{a(a +_{Int} 1)}{2}}$$

Fig. 5. An infinite proof tree under SYSTEP

Definition 5.3 (Symbolic Circular Coinduction) *Let G be a finite set of \mathcal{S} -derivable RL formulas. Then the set of rules $\text{SCC}(G)$ consists of SYSTEP together with*

$$\text{[circ]} \frac{\Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi \Rightarrow \varphi')}{\varphi \Rightarrow \varphi'} \mathcal{T} \models \varphi \rightarrow (\exists \text{var}(\varphi_c)) \varphi_c, \varphi_c \Rightarrow \varphi'_c \in G$$

The rule [circ] can be seen as a "big symbolic step" that accelerate the proving process. It does not increase the proving power of the system SYSTEP, but it

transform some infinite proof trees into finite ones, which can be built in finite time.

The following theorem, which we call *circularity principle*, states that when we add the circularity rule (and the circular reasoning that it allows) to **SYSTEM** it does not compromise soundness. The main reason is to start not with G , but with $\Delta_S(G)$, i.e., with the \mathcal{S} -derivatives of the formulas in G .

We shall be using the notation $\mathcal{S} \models G$ for $\mathcal{S} \models \varphi \Rightarrow \varphi'$ for all $\varphi \Rightarrow \varphi' \in G$.

Theorem 5.2 (Circularity Principle) *Assume \mathcal{S} total and that for each $\varphi_c \Rightarrow \varphi'_c \in G$, $\text{var}(\varphi'_c) \subseteq \text{var}(\varphi_c)$. If $\Delta_S(G) \subseteq \nu \widehat{\text{SCC}}(G)$ then $\mathcal{S} \models G$.*

Note that $\mathcal{S} \not\models \nu \widehat{\text{SCC}}(G)$ in general. For instance, for any arbitrary set of RL formulas G , each $\varphi \Rightarrow \varphi' \in G$ is in $\nu \widehat{\text{SCC}}(G)$ by applying the rule [circ]. Theorem 5.2 identifies a subset of proof trees under $\text{SCC}(G)$ that are sound w.r.t. $\mathcal{S} \models _$ (a proof tree for $\varphi \Rightarrow \varphi'$ under $\text{SCC}(G)$ is sound w.r.t. $\mathcal{S} \models _$ if $\mathcal{S} \models \varphi \Rightarrow \varphi'$): those for which the root is derived using the rule [der].

The advantage of using $\text{SCC}(G)$ is that it generates finite proof trees for a substantially larger set of RL formulas than **SYSTEM**. Its efficiency depends on the sets G given to it as input. This is quite similar to what happens when proving programs in Hoare logics, where the program specification needs additional information (under the form of loop invariants) in order to be successful.

Such a procedure can be used in an interactive way: first, it can be run it for proof trees of bounded height, and if it does not successfully terminates, one can analyse the proof trees built in order to guess some additional RL formulas, which, if added to G , increases the chance to find finite proof trees.

The finite proof tree under SCC that corresponds to the infinite proof tree T_2 under **SYSTEM** is represented in Fig. 6.

As final comments in this section, we note that, on the one hand, our 3-rule proof system SCC is substantially simpler than the original 8-rule proof system for RL given in [13]: a rule in SCC corresponds to many small reasoning steps in the original proof system. The downside is that we lose the theoretical *relative completeness* result of [13], which says that all valid RL formulas can (in principle) be proved. We prefer to focus on practical examples (shown in Section 7.1) to illustrate the usefulness of SCC . On the other hand, SCC is a strict generalisation of a RL verification procedure presented in [26]: that procedure can be seen as a strategy in SCC , in which implication is always applied before circularity, which is itself always applied before derivation.

$$\begin{array}{c}
\text{[impl]} \frac{}{(\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge b =_{\text{Int}} \frac{(a -_{\text{Int}} 1)a}{2} +_{\text{Int}} (s_0 +_{\text{Int}} a)} \\
\Rightarrow \\
(\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge b =_{\text{Int}} \frac{a(a +_{\text{Int}} 1)}{2} +_{\text{Int}} s_0 \\
\text{[circ]} \frac{}{\left\langle \begin{array}{l} \langle \text{while } (x \neq 0) \{s=s+x; \ x=x-1;\} P \rangle_k \\ \langle x \mapsto a -_{\text{Int}} 1 \ s \mapsto s_0 +_{\text{Int}} a \rangle_{\text{env}} \end{array} \right\rangle_{\text{cfg}} \wedge a \neq_{\text{Int}} 0} \\
\Rightarrow \\
(\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge b =_{\text{Int}} \frac{a(a +_{\text{Int}} 1)}{2} +_{\text{Int}} s_0 \\
\text{[der]} \frac{\dots}{\left\langle \begin{array}{l} \langle s=s+x; \ x=x-1; \text{while } (x \neq 0) \{s=s+x; \ x=x-1;\} P \rangle_k \\ \langle x \mapsto a \ s \mapsto s_0 \rangle_{\text{env}} \end{array} \right\rangle_{\text{cfg}} \wedge a \neq_{\text{Int}} 0} \\
\Rightarrow \\
(\exists b) \langle \langle P \rangle_k \langle x \mapsto 0 \ s \mapsto b \rangle_{\text{env}} \rangle_{\text{cfg}} \wedge b =_{\text{Int}} \frac{a(a +_{\text{Int}} 1)}{2} +_{\text{Int}} s_0
\end{array}$$

Fig. 6. The finite proof tree under SCC corresponding to the infinite proof tree T_2 in Fig. 5

6 Symbolic Execution via Language Transformation

As seen in the previous section, the derivative operation is essential both for symbolic execution and RL formula verification. In this section we show how the operation, currently defined as a matching, resp. reachability-logic formula transformer - i.e., $\Delta_S(\varphi) \triangleq \{(\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi)^{=?} \wedge \varphi_r \mid \varphi_l \Rightarrow \varphi_r \in \mathcal{S}\}$ for ML formulas φ , and, respectively $\Delta_S(\varphi \Rightarrow \varphi') \triangleq \{\varphi_1 \Rightarrow \varphi' \mid \varphi_1 \in \Delta_S(\varphi)\}$ for RL formulas $\varphi \Rightarrow \varphi'$ - can, under reasonable restrictions, be expressed by means of matching and rewriting. This is essential when implementing the proposed symbolic execution and verification techniques in a rewriting-based framework such as \mathbb{K} . We show that the language transformation technique proposed in [4] can be extended to the more general framework proposed here. The enhanced implementation is discussed in the next section.

Computing Derivatives by Matching

Consider a language definition $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$ over an ML signature $\Phi = (\Sigma, \Pi, Cfg)$. We assume a subsignature $(\Sigma^{Data}, \Pi^{Data})$ of (Σ, Π) consisting of all data sorts and their corresponding operations and predicates, e.g., integers, maps, trees, . . . depending on the language \mathcal{L} . We assume that the sort Cfg and the syntax of \mathcal{L} are not data, i.e., they are defined by operations in $\Sigma \setminus \Sigma^{Data}$.

We also assume that \mathcal{T} interprets the data sorts (those included in the subsignature Σ^{Data}) according to some Σ^{Data} -algebra \mathcal{D} (in which, for technical reasons, constants in Σ^{Data} are interpreted syntactically). Given the data model \mathcal{D} we construct a signature $\Sigma(\mathcal{D})$ by adding to Σ the elements of \mathcal{D} as constants of the corresponding sorts. Moreover, we assume that the model \mathcal{T} consists of equivalence classes of ground terms over $\Sigma(\mathcal{D})$, modulo a congruence \cong_A induced by a set of axioms A , e.g., any combination of associativity, commutativity and unity, for which there exists a *finitary and complete unification algorithm*, which moreover reduces to *matching* terms modulo the axioms in question.

In the sequel we shall mostly be concerned with pairs t, t' of terms, where

- t is a term in $T_{\Sigma(\mathcal{D}), Cfg}(Var^{Data})$, i.e., a basic pattern over Σ with variables in a set $Var^{Data} \subset Var$ of *data* variables, corresponding to basic patterns of configurations that are states in symbolic execution paths;
- t' is a term in $Tl_{\Sigma \setminus \Sigma^{Data}, Cfg}(Var)$, i.e., a *linear* basic pattern corresponding to the left-hand side of a rule in the semantics \mathcal{S} of the language of interest.

We shall see later in this section that the constraints on t and t' can be achieved by simple transformation on the rules \mathcal{S} defining the semantics of the language.

Definition 6.1 (A-unifier) *With the above notations, we say that two terms $t \in T_{\Sigma(\mathcal{D}), Cfg}(Var^{Data})$ and $t' \in Tl_{\Sigma \setminus \Sigma^{Data}, Cfg}(Var)$ are A -unifiable if there is $\rho : Var \rightarrow \mathcal{T}$ satisfying $t\rho \cong_A t'\rho$. The valuation ρ is an A -unifier of t, t' .*

Remark 6.1 *The congruence $t\rho \cong_A t'\rho$ in Definition 6.1 is just equality in the model \mathcal{T} , hence, it can also be written $t\rho = t'\rho$.*

Let us also define the A -matching between two terms of the above form.

Definition 6.2 (A-matcher) *With the above notations, given two terms $t \in T_{\Sigma(\mathcal{D}), Cfg}(Var^{Data})$ and $t' \in Tl_{\Sigma \setminus \Sigma^{Data}, Cfg}(Var)$, we say that a substitution $\sigma : var(t') \rightarrow T_{\Sigma(\mathcal{D})}(Var^{Data})$ is an A -matcher of t' onto t if $t \cong_A t'\sigma$.*

The next assumption requires that A -unification can be achieved by A -matching.

Assumption 6.1 We assume that for all terms $t \in T_{\Sigma(\mathcal{D}), Cfg}(Var^{Data})$ and $t' \in Tl_{\Sigma \setminus \Sigma^{Data}, Cfg}(Var)$, if t and t' are A -unifiable, then there is a finite set $\mu(t', t)$ of A -matchers of t' onto t , such that for each A -unifier ρ of t and t' , there are $\sigma_0 \in \mu(t', t)$ and a valuation $\rho' : Var \rightarrow \mathcal{T}$ such that $\rho = \sigma_0 \rho'$ ².

That is, each A -unifier is an instance of an A -matcher for the terms in question. The assumption holds under reasonable requirements³ (i.e., the axioms are *linear*, *regular*, and *non-collapsing*). Combinations of the usual associativity, commutativity, and unity (ACU) axioms typically satisfy these requirements, and \mathbb{K} language definitions intensively use these axioms (e.g., cells in configurations belong to ACU *bags*).

The second assumption restricts the class of RL formulas that can be used as program properties or as semantical rules to those that are actually useful in practice: left-hand sides are elementary patterns with linear basic patterns over the signature $\Sigma(\mathcal{D})$, whereas right-hand sides are quantified elementary patterns, such that there are no additional free variables in right-hand sides of formulas.

Assumption 6.2 We consider RL formulas $\varphi_1 \Rightarrow \varphi_2$ satisfying $var(\varphi_2) \subseteq var(\varphi_1)$, $\varphi_1 \triangleq \pi_1 \wedge \phi_1$ with $\pi_1 \in T_{\Sigma \setminus \Sigma^{Data}, Cfg}(Var)$, and $\varphi_2 \triangleq (\exists Y)\pi_2 \wedge \phi_2$.

Lemma 6.1 (ML Derivatives by Matching) Under the given assumptions, consider an RL formula $\varphi_1 \Rightarrow \varphi_2$ with $\varphi_1 \triangleq \pi_1 \wedge \phi_1$, $\varphi_2 \triangleq (\exists Y)\pi_2 \wedge \phi_2$, and $\varphi \triangleq (\exists X)\pi \wedge \phi$ such that $(X \cup var(\varphi)) \cap (Y \cup var(\varphi_1, \varphi_2)) = \emptyset$. Then,

$$\llbracket \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \rrbracket = \bigcup_{\sigma \in \mu(\pi_1, \pi)} \llbracket (\exists X, Y)\pi_2 \sigma \wedge \phi_1 \sigma \wedge \phi_2 \sigma \wedge \phi \rrbracket$$

In a nutshell the above lemma shows that semantically speaking, derivatives $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$ are certain ML formulas in which matchers (in the set $\mu(\pi_1, \pi)$) are present.

Computing Derivatives by Rewriting: Language Transformation

Lemma 6.1 shows how derivatives can equivalently be computed using matching. In this section we build on that result to show that derivatives can be computed by rewriting in a transformed language definition.

Consider a language definition $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$ over an ML signature $\Phi = (\Sigma, \Pi, Cfg)$. The transformation consists in creating a so-called *symbolic* language $\mathcal{L}^s = (\Phi^s, \mathcal{T}^s, \mathcal{S}^s)$ over a *symbolic* ML signature $\Phi^s = (\Sigma^s, \Pi^s, Cfg^s)$.

² $\rho_1 = \rho_2$ if for all $x \in Var$, $x\rho_1 = x\rho_2$, where "=" denotes equality in the model \mathcal{T} .

³ A proof of the assumption can be found in [44] (Th. 3: Unification by Matching).

Symbolic Signature. The algebraic signature Σ^s adds new sorts $Bool$ and Fol to Σ and, for each predicate $p \in \Pi_{s_1, \dots, s_n}$, a new operation $o_p : s_1 \times \dots \times s_n \rightarrow Bool$. Σ^s includes constructors for representing FOL formulas as terms of sort Fol . It also adds to Σ new sorts Id for identifiers and $IdList$ for lists of identifiers, with a (standard) equationally defined concatenation operation $_ , _ : IdList \times IdList \rightarrow IdList$. Finally, Σ^s includes a sort Cfg^s for symbolic configurations with its constructor $(\exists _)_ \wedge^s _ : IdList \times Cfg \times Fol \rightarrow Cfg^s$. The set Π^s consists of one single new predicate symbol: $sat : Fol \rightarrow Bool$.

Symbolic Model. The symbolic model \mathcal{T}^s interprets syntactically all operations in Σ^s except for the list concatenation operation, which is interpreted according to its (standard) equational definition. \mathcal{T}^s interprets the single predicate symbol $sat : Fol \rightarrow Bool$ as the satisfiability of FOL formulas: the interpretation of sat (conveniently, also denoted by sat) returns *true* iff the FOL formula interpreting the input of sat is satisfiable.

Symbolic Rules. Creating the set \mathcal{S}^s consists of the two following sub-steps:

- linearisation and Σ^{Data} -operation elimination: this transformation deals with the left-hand sides $\pi_1 \wedge \phi_1$ of RL formulas in \mathcal{S} , for which Assumption 6.2 requires $\pi_1 \in Tl_{\Sigma \setminus \Sigma^{Data}, Cfg}(Var)$, that is, π_1 is linear and does not contain any operations in Σ^{Data} (specifically, not even constants in Σ^{Data}). This is achieved by replacing in π_1 duplicated variables and subterms containing operations in Σ^{Data} with fresh variables, and by adding constraints to ϕ_1 that equate the newly introduced variables with the subterms they replaced.
- definition of the symbolic rules: for each rule $\pi_1 \wedge \phi_1 \Rightarrow (\exists Y)\pi_2 \wedge \phi_2 \in \mathcal{S}$, a rule

$$(\exists L)(\pi_1 \wedge^s \psi) \wedge sat(\phi_1 \wedge \psi) \Rightarrow (\exists L, Y)\pi_2 \wedge^s (\phi_2 \wedge \phi_1 \wedge \psi) \quad (11)$$

is created in \mathcal{S}^s , where ψ is a variable of the sort Fol , L is a variable of sort $IdList$, and $_ , _$ is the concatenation operation over $IdList$.

The symbolic language \mathcal{L}^s is now defined. Then we have a main result of this section, which says that derivatives can be computed by rewriting in \mathcal{L}^s .

Theorem 6.1 (Derivatives by Rewriting in \mathcal{L}^s) *Let \mathcal{S} be a set of RL formulas satisfying the assumptions of this section and $\varphi \triangleq (\exists X)\pi \wedge \phi$ be a quantified elementary pattern, such that $X \cup var(\varphi)$ is disjoint from the set of all variables (free or bound) occurring in \mathcal{S} . Then, $\llbracket \Delta_{\mathcal{S}}(\varphi) \rrbracket = \bigcup_{\varphi \Rightarrow_{\mathcal{S}^s} \varphi'} \llbracket \varphi' \rrbracket$.*

The theorem says that, semantically speaking, the derivation operation for configurations of a language \mathcal{L} amounts to computing successors according

to the transition relation in the symbolic language definition \mathcal{L}^s , which is computed by rewriting just as in any language definition in our framework.

The results in this section extend our own previous approach from [3] with new features such as existentially quantified patterns and rewriting modulo axioms. The extensions are theoretically nontrivial and are also important from a practical point of view. Specifically, RL formulas for program specifications (ranging from simple ones such as (2) to complex ones shown in the next section) do require quantifiers for expressiveness issues, and axioms are intensively used in \mathbb{K} language definitions as discussed earlier in the paper.

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.

7.1 Symbolic Execution within the \mathbb{K} Framework 3.4

We have integrated our symbolic execution framework in version 3.4 of the \mathbb{K} framework [42]. In \mathbb{K} , the definition of a language, say, \mathcal{L} , is compiled into a Maude [10] rewrite theory using the technique described in [44]. 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, as it is described in [4]. The main difference is that our new \mathbb{K} compiler includes the 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 semantical rules into symbolic rules of the form (11) according to the technique described in Section 6. The effect is that the compiled rewrite theory we obtain defines a symbolic semantics of \mathcal{L} in the sense of Section 4 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. A conservative approximation of the predicate *sat* is implemented using the \mathbb{K} 's interface to the Z3 SMT solver [16].

Given a \mathbb{K} language definition, our tool automatically generates its symbolic semantics. Thus, users that already have a \mathbb{K} language definition can sym-

```

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);
}

```

Fig. 7. CinK program: `init-arrays`

bolically execute their programs without having to change anything in them. Actually, symbolic execution using the transformed definition amounts to applying the set of rules `SYSTEM`.

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 7, 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 [2] 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 [2,4] since it uses conversions between arrays and pointers. We obtain the same result as [2,4] 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"
    -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 Reachability-Logic Verification

In this section we illustrate the usage of our verification prototype on the Knuth-Morris-Pratt [25] 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 whether a set of reachability formulas (goals) G holds, given language semantics \mathcal{S} , 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}^s of \mathcal{L} enriched with the rules from G . This new definition is used to perform symbolic execution of the patterns in left-hand sides of formulas in $\Delta_{\mathcal{S}}(G)$. Actually, symbolic execution using the enriched definition amounts to applying the set of rules $\text{SCC}(G)$. The tool gives priority to rules in G in order to accelerate the process of finding a proof.

7.3.1 Verifying the Knuth-Morris-Pratt string matching algorithm: KMP

The Knuth-Morris-Pratt algorithm [25] 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 [12].

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 $T = \text{ABADABCDA}$ and $P = \text{ABAC}$. 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 $\text{C} \neq \text{D}$.

The KMP algorithm uses a *prefix function* π , which, for each position j in P , returns the length of the longest proper prefix of the subpattern up to position j 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 in ABADABCDA , which saves a comparison of the characters preceding them, since the algorithm "already knows" that they are equal (here, they are both A).

```

/*@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-1-k+u]) & \
      (forall u:1..q-1)(pi[u]=Pi(u)) & \
      (forall j)((j > k & \ j in Pi*(q-1))
        implies p[j+1] != p[q]) & \
      k in Pi*(q-1) & \ 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-1-k+u]) & \
        (forall u:1..q-1)(pi[u]=Pi(u)) & \
        (forall j)((j > k & \ j in Pi*(q-1))
          implies p[j+1] != p[q]) & \
        k in Pi*(q-1) & \ 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)(p[u]=t[i-1-q+u]) & \
      (forall u:1..m)(pi[u]=Pi(u)) & \
      (exists v)((forall u:v+1..i-1) Theta(u)<m & \
        allOcc(Out,p,t,v))/\
      Theta(i)<=q+1 */
    while (q > 0 && p[q + 1] != t[i]) {
      /*@inv: 1<=m & \ 0<=q<m & \ 1 <= i <= n/\
        (forall u:1..q)(p[u]=t[i-1-q+u]) & \
        (forall u:1..m)(pi[u]=Pi(u)) & \
        (exists v)((forall u:v+1..i-1)
          Theta(u)<m & \ allOcc(Out,p,t,v))/\
        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) */

```

Fig. 8. The KMP algorithm annotated with pre-/post-conditions and invariants (syntactical sugar for RL formulas): prefix function (left) and the main function (right). In the annotations, Pi , Theta , Pi^* , and allOcc denote functions π and θ , the set π^* , and the predicate allOcc , respectively, which are defined axiomatically in an extension of CinK.

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 prefix 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 [12]. Another formal proof of the algorithm is given in [18] by using Why3 [19]. 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 [12]. We also prove that `KMP` finds *all* the occurrences of the pattern in the string, not only the first one. We use P_i to denote the prefix of P of size i (e.g., if $P \triangleq c_1c_2\dots c_m$ then $P_i \triangleq c_1\dots c_i$), and $P[i]$ to denote its i -th element (i.e., $P[i] \triangleq c_i$). Also, we denote by $P \sqsubset T$ the fact that P is a suffix of T .

Definition 7.1 *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_i which is also a suffix for P_i , for all $1 \leq i \leq m$; also, $\pi^*(i) \triangleq \{\pi(i), \pi(\pi(i)), \dots, 0\}$ (cf. [12])
- $\theta(i)$ is the length of the longest prefix of P that matches T on the final position i , i.e., $P_{\theta(i)} \sqsubset T_i$, where $1 \leq i \leq n$;
- $\text{allOcc}(Out, P, T, i)$ holds iff the list Out contains all the occurrences of P in T_i , where $1 \leq i \leq n$.

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

$$\left\langle \begin{array}{l} \langle \text{kmp_matcher}(p, t, m, n); \rangle_k \langle \cdot \rangle_{\text{out}} \\ \langle p \mapsto l_1 \quad t \mapsto l_2 \rangle_{\text{env}} \langle l_1 \mapsto P \quad l_2 \mapsto T \rangle_{\text{store}} \dots \end{array} \right\rangle_{\text{cfg}} \wedge n \geq 1 \wedge m \geq 1$$

$$\Rightarrow$$

$$\langle \langle \cdot \rangle_k \langle Out \rangle_{\text{out}} \langle \dots \rangle_{\text{env}} \langle \dots \rangle_{\text{store}} \dots \rangle_{\text{cfg}} \wedge \text{allOcc}(Out, P, T, n)$$

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. For instance, the above specification for `KMP` is generated from the following annotation:

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

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 $\langle \rangle_{\text{out}}$ 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 in the text.

Loops can be annotated with invariants using a syntax as follows:

```
while (COND) {
  //@inv: INV
  ...
}
```

where `INV` is the invariant (see Figure 8 for examples of invariants). For each annotated loop, the tool generates an RL formula which 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 six RL formulas.

Finally, 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 , π^* , θ , and *allOcc*, and we independently prove the following facts:

If P is a pattern of length m , π is its prefix function, and T a string of length n , then

- (1) for all $q \in \{1, \dots, m\}$, $\pi^*(q) = \{j \mid 0 \leq j < q \wedge P_j \sqsupset P_q\}$;⁴
- (2) for all $q \in \{1, \dots, m\}$, $0 \leq \pi(q) < q$;
- (3) for all $q \in \{1, \dots, m\}$, if $\pi(q) > 0$ then $\pi(q) - 1 \in \pi^*(q - 1)$;⁵
- (4) for all $q \in \{1, \dots, m\}$, for all $k \in \{1, \dots, m - 1\}$, for all $j \in \pi^*(q - 1)$,
if $(j \geq k \rightarrow P[j + 1] \neq P[q])$ then $\pi(q) \leq \pi(k) + 1$;
- (5) for all $q \in \{1, \dots, m\}$, if $u \in \pi^*(q)$ and $P_q \sqsupset T$ then $P_u \sqsupset T$;
- (6) for all $u, j \in \{1, \dots, m\}$, if $u < j$, $P_u \sqsupset T$, and $P_j \sqsupset T$ then $P_u \sqsupset P_j$.
- (7) for all $i \geq 1$, $\theta(i + 1) \leq \theta(i) + 1$.

In the case of `KMP`, we had to figure out first the needed functions and predicates specific to the problem domain to write the RL specifications and the invariants for loops. For this, we followed the insightful and very intuitive comments about the algorithm from [12]. Second, we identified the properties listed above using a trial and error process, which also required manual

⁴ Cf. Lemma 32.5 from [12]

⁵ Cf. Lemma 32.6 from [12]

labour. Essentially, this can be described in a few steps: use the tool to prove the formulas directly; in case of failure, get the translation of the formula for the solver and ask the solver to simplify the formula; then use the simplified formula and eventually some counterexamples returned by the solver to guess the needed property; check if this property holds; if it does, then append it to the set of axioms and reiterate the process until the tool finishes successfully.

The above example shows that the verification based on the circular coinduction technique proposed in Section 5.1 can be effective. The KMP example illustrate well that the program verification process also assumes the enrichment of the semantic domain, over which the language is defined, with additional reasoning in the problem domain. This can also be seen as a conservative extension of the language definition.

8 Conclusion and Future Work

We have presented a formal and generic framework for the symbolic execution of programs in languages definable in a matching-logic and term-rewriting setting. Symbolic execution is performed by applying the operation called *derivative*, which, under reasonable assumptions, can be computed by rewriting in a modified language definition. 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 one on the same path of instructions, and *precision*, meaning that each (feasible) symbolic execution has a concrete execution on the same path. Comparing with [4], these properties are set for a more general case and are expressed in terms of coinduction and mutual simulations. The incorporation of symbolic execution into a deductive system for program verification with respect to Reachability-Logic specifications is also presented. Finally, we present the implementation of a prototype tool based on the above theory 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 [28], 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 [40]). 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.

References

- [1] W. Ahrendt. The KeY tool. *Software and Systems Modeling*, 4:32–54, 2005.
- [2] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. Model checking linear programs with arrays. *Electronic Notes in Theoretical Computer Science*, 144(3):79–94, 2006.
- [3] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A generic framework for symbolic execution. In *6th International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 281–301. Springer Verlag, 2013. Also available as a technical report at <http://hal.inria.fr/hal-00766220/>.
- [4] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. Symbolic execution based on language transformation. *Computer Languages, Systems & Structures*, 44:48–71, 2015.
- [5] Mike Barnett, K. Rustan M. Leino, and Wolfram 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.
- [6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Programming Languages and Systems, Third Asian Symposium, APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.
- [7] Cristian Cadar, Daniel Dunbar, and Dawson 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.
- [8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

- [9] Edmund Clarke and Daniel 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.
- [10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso M. Oliet, Jos'e Meseguer, and Carolyn 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, 2007.
- [11] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Software Engineering Notes*, 26(5):142–151, 2001.
- [12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
- [13] Andrei Ștefănescu, Ștefan Ciobăcă, Radu Mereuță, Brandon M. Moore, Traian Florin Șerbănuță, and Grigore Roșu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA '14)*, volume 8560 of *Lecture Notes in Computer Science*, pages 425–440. Springer, July 2014.
- [14] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. *SIGPLAN Notes*, 48(4):329–342, March 2013.
- [15] Jonathan de Halleux and Nikolai Tillmann. Parameterized unit testing with Pex. In *Tests and Proofs, Second International Conference*, volume 4966 of *Lecture Notes in Computer Science*, pages 171–181. Springer, 2008.
- [16] Leonardo de Moura and Nikolaj 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 *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
- [17] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electronic Notes in Theoretical Computer Science*, 238(3):103–119, 2009.
- [18] Jean Christophe Filliâtre. Proof of kmp string searching algorithm. <http://toccata.lri.fr/gallery/kmp.en.html>.
- [19] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.
- [20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

- [21] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.
- [22] Bart Jacobs, Jan Smans, and Frank 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.
- [23] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew 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.
- [24] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
- [25] Donald E. Knuth, J.H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.
- [26] Dorel Lucanu, Vlad Rusu, Andrei Arusoaie, and David Nowak. Verifying Reachability-Logic Properties on Rewriting-Logic Specifications. In *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, Urbana Champaign, United States, September 2015.
- [27] Dorel Lucanu and Traian-Florin Șerbănuță. 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.
- [28] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
- [29] José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
- [30] Brandon Moore and Grigore Roșu. Program verification by coinduction. Technical Report <http://hdl.handle.net/2142/73177>, University of Illinois, February 2015.
- [31] Lawrence S. Moss and Norman Danner. On the foundations of corecursion. *Logic Journal of the IGPL*, 5(2):231–257, 1997.
- [32] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *International Conference on Automated Software Engineering, ASE'10*, pages 179–180. ACM, 2010.
- [33] Corina S. Păsăreanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.
- [34] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, 2009.

- [35] David A Ramos and Dawson 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.
- [36] John 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.
- [37] Camilo Rocha, José Meseguer, and César A. Muñoz. Rewriting modulo SMT and open system analysis. In *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, pages 247–262, 2014.
- [38] Grigore Roşu. Matching logic — extended abstract. In *Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA'15)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5–21, Dagstuhl, Germany, July 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [39] Grigore Roşu and Andrei Ştefănescu. From Hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.
- [40] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobăcă, and Brandon 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.
- [41] Grigore Roşu and Dorel Lucanu. Circular coinduction – a proof theoretical foundation. In *CALCO 2009*, volume 5728 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2009.
- [42] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
- [43] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012. Also available as technical report <http://hdl.handle.net/2142/33771>.
- [44] Vlad Rusu, Dorel Lucanu, Traian Florin Serbanuta, Andrei Arusoaiie, Andrei Ştefănescu, and Grigore Roşu. Language definitions as rewrite theories. *Journal of Logical and Algebraic Methods in Programming*, 85(1, Part 1):98 – 120, 2016. Rewriting Logic and its Applications.
- [45] Davide Sangiorgi. *An Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012. A preliminary version of Chapter 2 from which we adapt paterial can be found at http://www.cs.unibo.it/~sangiorgi/DOC_public/corsoFL.pdf.

- [46] Peter H. Schmitt and Benjamin Weiß. Inferring invariants by symbolic execution. In *VERIFY*, volume 259 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
- [47] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.
- [48] Traian-Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009.

A Proofs of Results from Section 4

Proof of Lemma 4.1, Page 23.

Let $\varphi_2 \triangleq \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi_1) \triangleq (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. Obviously $\varphi_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}}^s \varphi_2$. We now prove $(\diamond) (\gamma_2, \rho) \models (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. From $\gamma_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma_2$ we obtain a valuation η such that $(\gamma_1, \eta) \models \varphi_l$ and $(\gamma_2, \eta) \models \varphi_r$. Since variables of rules can always be renamed we can assume $\text{var}(\varphi_1) \cap \text{var}(\varphi_l, \varphi_r) = \emptyset$, thus, we can construct a valuation ρ' such that $\rho'|_{\text{var}(\varphi_1)} = \rho$ and $\rho'|_{\text{var}(\varphi_l, \varphi_r)} = \eta$, and then $(\gamma_2, \rho') \models \varphi_r$ and $(\gamma_1, \rho') \models \varphi_l \wedge \varphi_1$. Using Proposition 3.5 we obtain that $\rho' \models (\varphi_l \wedge \varphi_1)^{=?}$. Finally, using the definition of the \models relation we obtain $(\gamma_2, \rho') \models (\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$ and $(\gamma_2, \rho) \models (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$, which proves the lemma.

Proof of Theorem 4.1, Page 23. In this proof we use the following set of rules R :

$$\frac{}{\langle (\gamma, \rho), \varphi \rangle} (\gamma, \rho) \models \varphi \quad (\text{A.1})$$

$$\frac{\langle (\tau, \rho), \varphi \rangle}{\langle (\gamma_0 \Rightarrow_S \tau, \rho), \varphi_0 \rangle} (\gamma_0, \rho) \models \varphi_0, \varphi_0 \Rightarrow_S^s \varphi \quad (\text{A.2})$$

First, we show that $\nu \widehat{R}$ defines the set of all pairs of the form $\langle (\tau, \rho), \varphi \rangle$ such that (τ, ρ) starts from φ and there is a symbolic path that starts with φ and covers (τ, ρ) . For this we define coinductively a function sp over $\nu \widehat{R}$:

- $sp(\langle (\gamma, \rho), \varphi \rangle) = \varphi$
- $sp(\langle (\gamma_0 \Rightarrow_S \tau, \rho), \varphi_0 \rangle) = \varphi_0 \Rightarrow_S^s sp(\langle (\tau, \rho), \varphi \rangle)$, where φ is given by A.2.

It is easy to see that $sp(\langle (\tau, \rho), \varphi \rangle)$ is a symbolic path. Next, we coinductively prove that $sp(\langle (\tau, \rho), \varphi \rangle) \sqsupseteq (\tau, \rho)$ by showing that $Y \triangleq \{sp(\langle (\tau, \rho), \varphi \rangle), (\tau, \rho)\}$

$\langle (\tau, \rho), \varphi \rangle \in \nu \widehat{R}$ is backward closed w.r.t. $\widehat{(7, 8)}$, that is, $Y \subseteq \widehat{(7, 8)}(Y)$ (\clubsuit), where $\widehat{(7, 8)}(Y) = \{\langle \varphi, (\gamma, \rho) \rangle \mid (\gamma, \rho) \models \varphi\} \cup \{\langle \varphi_0 \Rightarrow_S^s \tau^s, (\gamma_0 \Rightarrow_S \tau, \rho) \rangle \mid (\gamma_0, \rho) \models \varphi_0, \langle \tau^s, (\tau, \rho) \rangle \in Y\}$

We choose an arbitrary $\langle sp(\langle (\tau', \rho'), \varphi' \rangle), (\tau', \rho') \rangle \in Y$. Note that $\langle (\tau', \rho'), \varphi' \rangle \in \nu \widehat{R}$. If $\langle (\tau', \rho'), \varphi' \rangle$ was obtained using (A.1), then τ' is an execution path consisting of a single concrete configuration, say $\tau' \triangleq \gamma'$, such that $(\gamma', \rho') \models \varphi'$. Moreover, $\varphi' = sp(\langle (\gamma', \rho'), \varphi' \rangle)$ and thus, $(\gamma', \rho') \models sp(\langle (\tau', \rho'), \varphi' \rangle)$ which implies $\langle sp(\langle (\tau', \rho'), \varphi' \rangle), (\tau', \rho') \rangle \in \widehat{(7, 8)}(Y)$.

On the other hand, if $\langle (\tau', \rho'), \varphi' \rangle$ was generated using (A.2) then there are γ', τ'' , and φ'' such that $\tau' \triangleq \gamma' \Rightarrow_S \tau''$, $(\gamma', \rho') \models \varphi'$, $\varphi' \Rightarrow_S^s \varphi''$, and $\langle (\tau'', \rho'), \varphi'' \rangle \in \nu \widehat{R}$. From $\langle (\tau'', \rho'), \varphi'' \rangle \in \nu \widehat{R}$ we have $\langle sp(\langle (\tau'', \rho'), \varphi'' \rangle), (\tau', \rho') \rangle \in Y$, which, combined with $(\gamma', \rho') \models \varphi'$ and the fact that $sp(\langle (\tau', \rho'), \varphi' \rangle) = \varphi' \Rightarrow_S^s sp(\langle (\tau'', \rho'), \varphi'' \rangle)$, implies $\langle sp(\langle (\tau', \rho'), \varphi' \rangle), (\tau', \rho') \rangle \in \widehat{(7, 8)}(Y)$.

Since $\langle sp(\langle (\tau', \rho'), \varphi' \rangle), (\tau', \rho') \rangle \in Y$ was arbitrarily chosen we obtain (\clubsuit).

Second, we show that $X = \{((\tau, \rho), \varphi) \mid (hd(\tau), \rho) \models \varphi\}$ (i.e, the set of all pairs $((\tau, \rho), \varphi)$ such that (τ, ρ) starts from φ) is backward closed w.r.t \widehat{R} , that is, $X \subseteq \widehat{R}(X)$, where $\widehat{R}(X)$ is: $\{((\gamma, \rho), \varphi) \mid (\gamma, \rho) \models \varphi\} \cup \{((\gamma_0 \Rightarrow_S \tau, \rho), \varphi_0) \mid (\gamma_0, \rho) \models \varphi, \varphi_0 \Rightarrow_S^s \varphi, (hd(\tau), \rho) \models \varphi\}$.

Consider $((\tau, \rho), \varphi) \in X$. We distinguish the following cases:

- $\tau = \gamma$, γ - irreducible. Since $((\tau, \rho), \varphi) \in X$ we have $(hd(\tau), \rho) \models \varphi$ which becomes $(\gamma, \rho) \models \varphi$. Using (A.1) we obtain $((\tau, \rho), \varphi) \in \widehat{R}(X)$.
- $\tau = \gamma_0 \Rightarrow_S \tau'$. From $((\tau, \rho), \varphi) \in X$ we obtain $(hd(\tau), \rho) \models \varphi$, that is, $(\gamma_0, \rho) \models \varphi$. On the other hand, $\gamma_0 \Rightarrow_S hd(\tau')$. By Lemma 4.1, there is φ' such that $(hd(\tau'), \rho) \models \varphi'$ and $\varphi \Rightarrow_S^s \varphi'$. Since $(hd(\tau'), \rho) \models \varphi'$ then $((\tau', \rho), \varphi') \in X$. Thus, using (A.2) we obtain $((\tau, \rho), \varphi) \in \widehat{R}(X)$.

Using the coinduction principle we obtain $X \subset \nu \widehat{R}$. Thus, we started with an arbitrary pair $((\tau, \rho), \varphi)$ such that (τ, ρ) starts from φ and proved that the pair also has the property that there is a symbolic path that starts with φ and covers (τ, ρ) , which proves the lemma.

Proof of Lemma 4.2, Page 24.

From $\varphi_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}}^s \varphi_2$ we have $\varphi_2 = \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi_1) \triangleq (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. From $(\gamma_2, \rho) \models (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$ we obtain that there is $\rho'' : Var \rightarrow \mathcal{T}$ such that $(\gamma_2, \rho'') \models (\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$ and $\rho''|_{Var \setminus var(\varphi_l, \varphi_r)} = \rho|_{Var \setminus var(\varphi_l, \varphi_r)}$. From $(\gamma_2, \rho'') \models (\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$ we obtain in particular $\rho'' \models (\varphi_l \wedge \varphi_1)^{=?}$, hence, by Proposition 3.5 there exists γ_1 such that $(\gamma_1, \rho'') \models \varphi_l \wedge \varphi_1$. Since we can assume w.r.g. $var(\varphi_1) \cap var(\varphi_l, \varphi_r) = \emptyset$ we obtain in

particular $\rho''|_{\text{var}(\varphi_1)} = \rho|_{\text{var}(\varphi_1)}$, hence, $(\gamma_1, \rho) \models \varphi_1$. We choose this γ_1 to be the configuration whose existence is stated in the conclusion of our lemma.

To prove the lemma there remains to show $(\spadesuit) \gamma_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma_2$. From $(\gamma_1, \rho'') \models \varphi_l \wedge \varphi_1$ we obtain in particular $(\gamma_1, \rho'') \models \varphi_l$, and from $(\gamma_2, \rho'') \models (\varphi_l \wedge \varphi_1) \stackrel{?}{=} \wedge \varphi_r$ we obtain in particular $(\gamma_2, \rho'') \models \varphi_r$, which proves (\spadesuit) and the lemma.

Proof of Theorem 4.2, Page 24. Let R' denote the following set of rules:

$$\frac{}{\varphi} (\exists \gamma, \rho)(\gamma, \rho) \models \varphi, \quad (\text{A.3})$$

$$\frac{\tau^s}{\varphi \Rightarrow_{\mathcal{S}}^s \tau^s} \text{hd}(\tau^s) \text{ satisfiable}, (\forall \gamma', \rho)(\gamma', \rho) \models \text{hd}(\tau^s) \rightarrow (\exists \gamma)(\gamma, \rho) \models \varphi \wedge \gamma \Rightarrow_{\mathcal{S}} \gamma' \quad (\text{A.4})$$

We first prove $(\diamond) : \mu R \subseteq \mu R'$, where \mathcal{R} is the set of rules from Definition 4.1. We prove that $X' = \mu R'$ is closed w.r.t. \widehat{R} , i.e., $\widehat{R}(X') \subseteq X'$ and apply the Induction principle. Elements of $\widehat{R}(X')$ generated by the 1st rule of R consist of one (satisfiable) ML formula, hence, they are also obviously in X' . Elements of $\widehat{R}(X')$ generated by the 2nd rule of R are of the form $\tau'^s = \varphi \Rightarrow_{\mathcal{S}}^s \tau^s$ where φ is \mathcal{S} -derivable and $\text{hd}(\tau^s) \in \Delta_{\mathcal{S}}(\varphi)$ is satisfiable, cf. Remark 4.1. The constraint $(\forall \gamma')(\gamma', \rho) \models \text{hd}(\tau^s) \rightarrow (\exists \gamma)(\gamma, \rho) \models \varphi \wedge \gamma \Rightarrow_{\mathcal{S}} \gamma'$ holds due to Lemma 4.2. Hence, the arbitrarily chosen $\tau'^s \in X'$ generated by the 2nd rule of R satisfies the 2nd rule of R' , which implies $\tau'^s \in \widehat{R}(X')$, and we can conclude $\widehat{R}(X') \subseteq X'$. The proof of (\diamond) is now complete.

Next we prove (\spadesuit) that the set $Y = \{\tau^s \in \mu R \mid (\exists \tau, \rho) \tau^s \sqsupseteq (\tau, \rho)\}$ is closed w.r.t. \widehat{R}' , i.e., $\widehat{R}'(Y) \subseteq Y$. Elements of $\widehat{R}'(Y)$ generated by the 1st rule of R' consist of one (satisfiable) ML formula φ , which covers the pair (γ, ρ) satisfying φ , hence, such φ are also obviously in Y . Elements of $\widehat{R}'(Y)$ generated by the 2nd rule of R' have the form $\tau'^s = \varphi \Rightarrow_{\mathcal{S}}^s \tau^s$ where $\tau^s \in Y$ (hence, τ^s covers some (τ, ρ)), $\text{hd}(\tau^s)$ is satisfiable, and $(\forall \gamma')(\gamma', \rho) \models \text{hd}(\tau^s) \rightarrow (\exists \gamma)(\gamma, \rho) \models \varphi \wedge \gamma \Rightarrow_{\mathcal{S}} \gamma'$ holds (from the condition of the 2nd rule of R'). We have $(\text{hd}(\tau), \rho) \models \text{hd}(\tau^s)$ from $\tau^s \sqsupseteq (\tau, \rho)$ thus, there exists γ such that $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_{\mathcal{S}} \text{hd}(\tau)$, which proves $\tau'^s = \varphi \Rightarrow_{\mathcal{S}}^s \tau^s \sqsupseteq (\gamma \Rightarrow_{\mathcal{S}} \tau, \rho)$ and thus the arbitrarily chosen $\tau'^s \in Y$ generated by the 2nd rule of R' is in Y .

We conclude $\widehat{R}'(Y) \subseteq Y$, and thus $\mu R' \subseteq Y$. From (\diamond) we obtain $\mu R \subseteq Y$, i.e., any symbolic path τ^s satisfies $(\exists \tau, \rho) \tau^s \sqsupseteq (\tau, \rho)$, which proves the theorem.

B Proofs of Results from Section 5

We first introduce a definition and prove a lemma.

Definition B.1 Consider the following rule:

$$\frac{}{\langle \varphi, (\tau, \rho) \rangle} (hd(\tau), \rho) \models \varphi \quad (\text{B.1})$$

We write $\tau^s \succeq (\tau, \rho)$ for $\langle \tau^s, (\tau, \rho) \rangle \in \nu(\widehat{\text{B.1, 8}})$, where (B.1, 8) is the system of the ground rules defined by (B.1) and (8).

If $\tau^s \succeq (\tau, \rho)$ then the concrete execution path can be longer than the symbolic one. We obviously have that $\tau^s \sqsupseteq (\tau, \rho)$ implies $\tau^s \succeq (\tau, \rho)$. We call \succeq the *partial cover* relation, by contrast to the *cover* relation \sqsupseteq .

Lemma B.1 For all (τ, ρ) and $\varphi \Rightarrow \varphi'$, if there exists τ^s such that $\tau^s \models \varphi \Rightarrow \varphi'$ and $\tau^s \succeq (\tau, \rho)$ then $(\tau, \rho) \models \varphi \Rightarrow \varphi'$.

Proof We proceed by coinduction by showing that the set $X = \{\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle \mid (\exists \tau^s) \tau^s \succeq (\tau, \rho), \tau^s \models \varphi \Rightarrow \varphi'\}$ is backwards closed w.r.t. $(\widehat{3, 4})$. The set X includes all the pairs $\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle$ such that there is a symbolic path τ^s which partially covers (τ, ρ) and $\tau^s \models \varphi \Rightarrow \varphi'$.

We arbitrarily choose $\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle \in X$. Then, there is τ^s such that $\tau^s \succeq (\tau, \rho)$ and $\tau^s \models \varphi \Rightarrow \varphi'$. We distinguish the following cases:

- (1) $\tau^s = \varphi_0$. Since $\tau^s \models \varphi \Rightarrow \varphi'$ then $\mathcal{T} \models \varphi_0 \rightarrow \varphi \wedge \varphi'$ (cf. Definition 5.1). Also, from $\tau^s \succeq (\tau, \rho)$, that is, $\varphi_0 \succeq (\tau, \rho)$, we obtain $(hd(\tau), \rho) \models \varphi_0$ by (B.1), and thus, $(hd(\tau), \rho) \models \varphi \wedge \varphi'$ (cf. Definition 3.8). Hence $\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle \in (\widehat{3, 4})(X)$ by using (3).
- (2) $\tau^s = \varphi_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}} \tau'^s$, where $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$. From $\tau^s \models \varphi \Rightarrow \varphi'$ we obtain $\mathcal{T} \models \varphi_0 \rightarrow \varphi$ and $\tau'^s \models \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi')$ (\spadesuit) (cf. Definition 3.17). On the other hand, from $\tau^s \succeq (\tau, \rho)$, we obtain $\tau = \gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}} \tau'$, $(\gamma_0, \rho) \models \varphi_0$, and $\tau'^s \succeq (\tau', \rho)$ (\diamond) (cf. Definition 4.2). Using (\spadesuit) and (\diamond) we have that $((\tau', \rho), \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi')) \in X$, which, together with $(\gamma_0, \rho) \models \varphi_0 \wedge \varphi$ ensures that $\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle \in (\widehat{3, 4})(X)$ by using (4).

Since $\langle (\tau, \rho), \varphi \Rightarrow \varphi' \rangle \in X$ was chosen arbitrarily it follows that $X \subseteq (\widehat{3, 4})(X)$, i.e., X is backwards closed w.r.t. $(\widehat{3, 4})$. By applying the coinduction principle we obtain $X \subseteq \nu(\widehat{3, 4})$. \square

Proof of Theorem 5.1, Page 25. We start by recalling the theorem:

If \mathcal{S} is total then $\mathcal{S} \models \nu \widehat{\text{SYSTEM}}$.

Let P be a proof tree of an RL formula under SYSTEM. We define a function $f(\varphi \Rightarrow \varphi', (\tau, \rho))$, for $\varphi \Rightarrow \varphi' \in P$ and (τ, ρ) a complete execution path

starting from φ , that computes a symbolic execution path that partially covers (τ, ρ) and satisfies $\varphi \Rightarrow \varphi'$:

- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi$ if $\mathcal{T} \models \varphi \rightarrow \varphi'$.
Since (τ, ρ) starts from φ and $\mathcal{T} \models \varphi \rightarrow \varphi'$, we obtain $(hd(\tau), \rho) \models \varphi \wedge \varphi'$;
- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi \Rightarrow_{\mathcal{S}}^s f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi', (\tau', \rho)))$ if φ is \mathcal{S} -derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^p hd(\tau')$.
Note that γ_0 and $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ exists because \mathcal{S} is total and φ is \mathcal{S} -derivable. If there is more than one rule $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ that matches the step $\gamma_0 \Rightarrow_{\mathcal{S}}^p hd(\tau')$, then f chooses arbitrarily one of them. We have $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi') \in P$, and (τ', ρ) is a complete execution path starting from $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$. Since $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$ starts from φ we have $(\gamma_0, \rho) \models \varphi$. Also, by Definition 3.11 we obtain $(\gamma_0, \rho) \models \varphi_1$, and thus, $(\gamma_0, \rho) \models \varphi \wedge \varphi_1$.

In order to show that $f(\varphi \Rightarrow \varphi', (\tau, \rho))$ satisfies $\varphi \Rightarrow \varphi'$ we prove that the set $X = \{\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \mid \varphi \Rightarrow \varphi' \in P, (\tau, \rho) \text{ starts from } \varphi\}$ is backward closed w.r.t. $\widehat{(9, 10)}$. Let $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle$ be an arbitrarily chosen pair in X . We distinguish the following cases:

- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi$ and $\mathcal{T} \models \varphi \rightarrow \varphi'$. In this case, since $\mathcal{T} \models \varphi \rightarrow \varphi$ we get $\mathcal{T} \models \varphi \rightarrow \varphi \wedge \varphi'$ and we obtain $\langle \varphi, \varphi \Rightarrow \varphi' \rangle \in \widehat{(9, 10)}(X)$ by (9), i.e. $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in \widehat{(9, 10)}(X)$.
- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi \Rightarrow_{\mathcal{S}}^s f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi', (\tau', \rho)))$, φ is \mathcal{S} -derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^p hd(\tau')$. We have $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi', (\tau', \rho))), \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi') \rangle \in X$ because $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi') \in P$ by the definition of proof trees under **SYSTEM** and (τ', ρ) is a complete execution path starting from $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$ by the definition of $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^p hd(\tau')$ and $(\gamma_0, \rho) \models \varphi \wedge \varphi_1$. Since $\mathcal{T} \models \varphi \rightarrow \varphi$ we obtain $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in \widehat{(9, 10)}(X)$ by (10).

Now we have that X is backward closed w.r.t. to $\widehat{(9, 10)}$. Using the coinduction principle we obtain $X \subseteq \nu \widehat{(9, 10)}$, i.e. $f(\varphi \Rightarrow \varphi', (\tau, \rho)) \models \varphi \Rightarrow \varphi'$ for each $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in X$.

Next, we show that $f(\varphi \Rightarrow \varphi', (\tau, \rho))$ partially covers (τ, ρ) in order to apply Lemma B.1 and prove our theorem. We prove that the set $Y = \{\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle \mid \varphi \Rightarrow \varphi' \in P, (\tau, \rho) \text{ starts from } \varphi\}$ is backwards closed w.r.t. $\widehat{(B.1, 8)}$. We arbitrarily choose $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle$ in Y . We distinguish the following cases:

- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi$ and $\mathcal{T} \models \varphi \rightarrow \varphi'$. Since $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle$ is in Y it follows that (τ, ρ) starts from φ , i.e., $(hd(\tau), \rho) \models \varphi$. We obtain $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in \widehat{(B.1, 8)}(Y)$ by (B.1).
- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi \Rightarrow_{\mathcal{S}}^s f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi', (\tau', \rho)))$, φ is \mathcal{S} -derivable,

$\tau = \gamma_0 \Rightarrow_S \tau'$, and $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^{\rho} hd(\tau')$. We prove that $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi', (\tau', \rho)) \in Y$ in a similar way we proceeded for showing that $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi', (\tau', \rho)), \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi' \in X$. From $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle \in Y$ we obtain (τ, ρ) starts from φ , i.e., $(\gamma_0, \rho) \models \varphi$. By using (8) it follows that $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle \in (\widehat{B.1, 8})(Y)$.

We have $Y \subseteq (\widehat{B.1, 8})(Y)$, and by the coinduction principle we obtain $Y \subseteq \nu(\widehat{B.1, 8})$, i.e. $f(\varphi \Rightarrow \varphi', (\tau, \rho)) \succeq (\tau, \rho)$ for each $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle \in Y$. Now f is completely defined and has the desired properties.

If $\varphi \Rightarrow \varphi' \in \nu \widehat{\text{SYSTEM}}$ then there is a proof tree P of $\varphi \Rightarrow \varphi'$ under **SYSTEM**. If (τ, ρ) is a complete execution path starting from φ , then $f(\varphi \Rightarrow \varphi', (\tau, \rho))$ describes a symbolic path that covers (τ, ρ) and satisfies $\varphi \Rightarrow \varphi'$, which, by Lemma B.1, implies that (τ, ρ) satisfies $\varphi \Rightarrow \varphi'$. Since (τ, ρ) is arbitrarily chosen, it follows that $\mathcal{S} \models \varphi \Rightarrow \varphi'$ by Definition 3.17.

Proof of Theorem 5.2, Page 27. We start with some intermediary lemmas, then prove the theorem, which we re-state here for the sake of clarity:

Assume \mathcal{S} total and that for each $\varphi_c \Rightarrow \varphi'_c \in G$, $\text{var}(\varphi'_c) \subseteq \text{var}(\varphi_c)$. If $\Delta_{\mathcal{S}}(G) \subseteq \nu \widehat{\text{SCC}}(G)$ then $\mathcal{S} \models G$.

Lemma B.2 *For all ML formulas φ and φ' , and for all RL formulas $\varphi_l \Rightarrow \varphi_r$, if $\mathcal{T} \models \varphi \rightarrow \varphi'$ then $\mathcal{T} \models \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi) \rightarrow \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi')$.*

Proof Consider an arbitrary (γ', ρ) such that $(\gamma', \rho) \models \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi)$. Using Lemma 4.2 we obtain a configuration γ such that $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma'$. From $\mathcal{T} \models \varphi \rightarrow \varphi'$ we obtain $(\gamma, \rho) \models \varphi'$, which, together with $\gamma \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma'$ gives us using Lemma 4.1 the ML formula $\varphi'' \triangleq \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi')$ such that $(\gamma', \rho) \models \varphi''$, which proves the lemma. \square

We write $\tau^s \sqsupseteq \tau$ iff there exists $\rho : \text{Var} \rightarrow \mathcal{T}$ such that $\tau^s \sqsupseteq (\tau, \rho)$. The \sqsupseteq relation can be extended to symbolic paths, too:

Definition B.2 *Let τ^s and τ'^s be two symbolic execution paths. Then $\tau^s \sqsupseteq \tau'^s$ iff $\tau'^s \sqsupseteq \tau$ implies $\tau^s \sqsupseteq \tau$, for all execution paths τ .*

The following technical lemma states a property of (one-step) symbolic execution paths regarding the \sqsupseteq relation:

Lemma B.3 *Let $\varphi'_1 \Rightarrow_S^s \varphi'_2$ be a one-step symbolic execution path and φ_1 an \mathcal{S} -derivable ML formula. If $\mathcal{T} \models \varphi'_1 \rightarrow (\exists \text{var}(\varphi_1))\varphi_1$ then there exists φ_2 such that $\varphi_1 \Rightarrow_S^s \varphi_2 \sqsupseteq \varphi'_1 \Rightarrow_S^s \varphi'_2$ and $\mathcal{T} \models \varphi'_2 \rightarrow \varphi_2$.*

Proof We assume w.l.o.g. that $\text{var}(\varphi_1) \cap \text{var}(\varphi'_1, \varphi'_2) = \emptyset$ (variables in formulas can be renamed if necessary).

By Definition 4.1 we have on the one hand that $\varphi'_2 \in \Delta_{\mathcal{S}}(\varphi'_1)$, i.e., there is a rule $\alpha \triangleq \varphi_l \Rightarrow \varphi_r \in \mathcal{S}$ such that $\varphi'_2 \triangleq (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi'_1)^{=?} \wedge \varphi_r$ (cf. Definition 3.14), and on the other hand that φ'_2 is satisfiable since φ'_1 is \mathcal{S} -derivable as required by Definition 4.1.

Let us choose $\varphi_2 \triangleq (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. Obviously, $\varphi_2 \in \Delta_{\mathcal{S}}(\varphi_1)$. In order to show that $\varphi_1 \Rightarrow_{\mathcal{S}}^5 \varphi_2$ is indeed a symbolic execution path (cf. Definition 4.1) we have to prove that φ_2 is a symbolic execution path itself, that is, that φ_1 is \mathcal{S} -derivable and φ_2 is satisfiable. Note that for a symbolic execution path consisting of one transition the two above conditions are equivalent.

Hence, we prove that φ_2 is satisfiable. For this, we prove the following result:

For all (γ_2, ρ) , if $(\gamma_2, \rho) \models \varphi'_2$ then $(\gamma_2, \rho) \models \varphi_2$ (\spadesuit).

Let (γ_2, ρ) be an arbitrary pair such that $(\gamma_2, \rho) \models \varphi'_2$, that is, $(\gamma_2, \rho) \models (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi'_1)^{=?} \wedge \varphi_r$. By Definition 3.8, there is a valuation ρ' with $x\rho' = x\rho$ for all $x \notin \text{var}(\varphi_l, \varphi_r)$ such that $(\gamma_2, \rho') \models (\varphi_l \wedge \varphi'_1)^{=?} \wedge \varphi_r$. Hence, we have $(\gamma_2, \rho') \models (\varphi_l \wedge \varphi'_1)^{=?}$ and $(\gamma_2, \rho') \models \varphi_r$. Since $(\varphi_l \wedge \varphi'_1)^{=?}$ is a FOL formula then we have $\rho' \models (\varphi_l \wedge \varphi'_1)^{=?}$. By Proposition 3.5, there is γ_0 such that $(\gamma_0, \rho') \models (\varphi_l \wedge \varphi'_1)$, i.e., $(\gamma_0, \rho') \models \varphi_l$ and $(\gamma_0, \rho') \models \varphi'_1$.

From the hypothesis, $\mathcal{T} \models \varphi'_1 \rightarrow (\exists \text{var}(\varphi_1))\varphi_1$, which implies that $(\gamma_0, \rho') \models (\exists \text{var}(\varphi_1))\varphi_1$, where from we obtain a valuation ρ'' such that $x\rho'' = x\rho'$ for all $x \notin \text{var}(\varphi_1)$ and $(\gamma_0, \rho'') \models \varphi_1$. Since $\text{var}(\varphi_1) \cap \text{var}(\varphi'_1, \varphi'_2) = \emptyset$ we can choose ρ' (and hence ρ) such that $x\rho'' = x\rho'$ for $x \in \text{var}(\varphi_1)$ and we obtain $(\gamma_0, \rho') \models \varphi_1$.

It follows that $(\gamma_0, \rho') \models (\varphi_l \wedge \varphi_1)$ and (again, by Proposition 3.5) $\rho' \models (\varphi_l \wedge \varphi_1)^{=?}$. Since $(\gamma_2, \rho') \models \varphi_r$ then $(\gamma_2, \rho') \models (\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. Now, we have a valuation ρ' with $x\rho' = x\rho$ for all $x \notin \text{var}(\varphi_l, \varphi_r)$ such that $(\gamma_2, \rho') \models (\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. By Definition 3.8 we obtain $(\gamma_2, \rho) \models (\exists \text{var}(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. Hence, $(\gamma_2, \rho) \models \varphi_2$. Note that (γ_2, ρ) was arbitrarily chosen, and thus we have the satisfaction relation $\mathcal{T} \models \varphi'_2 \rightarrow \varphi_2$ (\clubsuit).

Also, φ'_2 -satisfiable implies that there are γ'_2 and ρ'_2 such that $(\gamma'_2, \rho'_2) \models \varphi'_2$. Using \clubsuit we obtain $(\gamma'_2, \rho'_2) \models \varphi_2$ which implies that φ_2 is satisfiable too, which proves (\spadesuit).

Thus, we conclude that $\varphi_1 \Rightarrow_{\mathcal{S}}^5 \varphi_2$ is indeed a symbolic execution path.

There remains to prove $\varphi_1 \Rightarrow_{\mathcal{S}}^5 \varphi_2 \sqsubseteq \varphi'_1 \Rightarrow_{\mathcal{S}}^5 \varphi'_2$. Let $\tau^s \triangleq \varphi_1 \Rightarrow_{\mathcal{S}}^5 \varphi_2$ and $\tau'^s \triangleq \varphi'_1 \Rightarrow_{\mathcal{S}}^5 \varphi'_2$. We have to prove that for every execution path $\tau \triangleq \gamma_1 \Rightarrow_{\mathcal{S}} \gamma_2$,

if $\tau'^s \sqsupseteq \tau$ then $\tau^s \sqsupseteq \tau$ (cf. Definition B.2). From $\tau'^s \sqsupseteq \tau$ and Definition 4.2 we obtain a valuation ρ such that $(\gamma_1, \rho) \models \varphi'_1$ and $(\gamma_2, \rho) \models \varphi'_2$. From $\mathcal{T} \models \varphi'_1 \rightarrow (\exists \text{var}(\varphi_1))\varphi_1$ and $(\gamma_1, \rho) \models \varphi'_1$ we obtain a valuation ρ' such that $(\gamma_1, \rho') \models \varphi_1$ and $x\rho = x\rho'$ for all $x \notin \text{var}(\varphi_1)$.

Let $\rho'' : \text{Var} \rightarrow \mathcal{T}$ be a valuation defined as follows: $x\rho'' = x\rho$, for all $x \notin \text{var}(\varphi_1)$ and $x\rho'' = x\rho'$ for all $x \in \text{var}(\varphi_1)$. From $(\gamma_1, \rho') \models \varphi_1$ we obtain $(\gamma_1, \rho'') \models \varphi_1$ because ρ'' coincides with ρ' on $\text{var}(\varphi_1)$. On the other hand, from $(\gamma_2, \rho) \models \varphi'_2$ and the fact that $\text{var}(\varphi_1) \cap \text{var}(\varphi'_1, \varphi'_2) = \emptyset$ we obtain $(\gamma_2, \rho'') \models \varphi'_2$ since ρ'' coincides with ρ on variables not in $\text{var}(\varphi_1)$, particularly on $\text{var}(\varphi'_2)$. From $(\gamma_2, \rho'') \models \varphi'_2$ and (\spadesuit) we obtain $(\gamma_2, \rho'') \models \varphi_2$.

Thus, $\varphi_1 \Rightarrow_{\mathcal{S}}^s \varphi_2 \sqsupseteq (\tau, \rho'')$, i.e., $\varphi_1 \Rightarrow_{\mathcal{S}}^s \varphi_2 \sqsupseteq \tau$, which proves the lemma. \square

Corollary B.1 *Let $\varphi'_1 \Rightarrow_{\mathcal{S}}^s \varphi'_2$ be a one-step symbolic execution path and φ_1 an \mathcal{S} -derivable ML formula. If $\mathcal{T} \models \varphi'_1 \rightarrow \varphi_1$ then there exists φ_2 such that $\varphi_1 \Rightarrow_{\mathcal{S}}^s \varphi_2 \sqsupseteq \varphi'_1 \Rightarrow_{\mathcal{S}}^s \varphi'_2$ and $\mathcal{T} \models \varphi'_2 \rightarrow \varphi_2$.*

Remark B.1 *In the rest of this section we assume the hypotheses of our theorem: \mathcal{S} is total; for each $\varphi_c \Rightarrow \varphi'_c \in G$, $\text{var}(\varphi'_c) \subseteq \text{var}(\varphi_c)$; and $\Delta_{\mathcal{S}}(G) \subseteq \nu \widehat{\text{SCC}}(G)$.*

Lemma B.4 $G \subseteq \nu \widehat{\text{SCC}}(G)$.

Proof Each $\varphi'' \Rightarrow \varphi'_c \in \Delta_{\mathcal{S}}(\varphi_c \Rightarrow \varphi'_c)$ has a proof tree $P_{\varphi'' \Rightarrow \varphi'_c}$ under SCC, for all $\varphi_c \Rightarrow \varphi'_c \in G$. It follows that there is a proof tree $P_{\varphi_c \Rightarrow \varphi'_c}$ under SCC for each $\varphi_c \Rightarrow \varphi'_c \in G$ obtained by applying the rule [der]. Hence $G \subseteq \nu \widehat{\text{SCC}}(G)$. \square

Lemma B.5 *Let $\varphi_c \Rightarrow \varphi'_c \in G$. If φ is \mathcal{S} -derivable and $\mathcal{T} \models \varphi \rightarrow (\exists \text{var}(\varphi_c))\varphi_c$ then there is a proof tree P for $\varphi \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$ under SCC(G).*

Proof We show that each rule applied in the construction of $P_{\varphi_c \Rightarrow \varphi'_c}$ (whose existence is known by Lemma B.4) can be used to construct P as well, provided that the left-hand side of the current node in P is satisfiable. Each time the same rule is applied for two corresponding nodes in P respectively $P_{\varphi_c \Rightarrow \varphi'_c}$, there is a bijection between the children of the two nodes that preserves the way each child is obtained.

At the end we shall obtain an injective tree homomorphism from P to $P_{\varphi_c \Rightarrow \varphi'_c}$ that maps the root of P into the root of $P_{\varphi_c \Rightarrow \varphi'_c}$ and each child of a node x in P into its correspondent given by the bijection between the children of x and the children of its homomorphic image. The only exception when different rules are applied is that in which for a node in P the inference rule [impl] is applied and for its homomorphic image [der] is applied; in those cases the

corresponding node in P has no children.

Since φ is \mathcal{S} -derivable, the rule [der] can be applied to the root of the proof tree $P \varphi \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$. Recall (cf. proof of Lemma B.4) that the first rule applied in the construction of $P_{\varphi_c \Rightarrow \varphi'_c}$ is [der] as well.

Thus, for the roots we may apply the same rule in \mathcal{S} for building P and $P_{\varphi_c \Rightarrow \varphi'_c}$. Moreover, if a child $\varphi' \Rightarrow \varphi'_c$ of $\varphi_c \Rightarrow \varphi'_c$ is the homomorphic image of the child $\varphi'' \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$ of $\varphi \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$, we have $\mathcal{T} \models \varphi'' \rightarrow \varphi'$ by Lemma B.3. We shall see that this relation will be maintained between the nodes of P and their homomorphic image in $P_{\varphi_c \Rightarrow \varphi'_c}$, except the roots (where $\mathcal{T} \models \varphi \rightarrow (\exists \text{var}(\varphi_c)\varphi_c)$, which allowed us above to apply Lemma B.3).

For the rest of the trees we distinguish the following cases (we assume that $\varphi' \Rightarrow \varphi'_c$ is the homomorphic image of $\varphi'' \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$ in each case):

- the rule [impl] is applied for $\varphi' \Rightarrow \varphi'_c$, hence $\mathcal{T} \models \varphi' \rightarrow \varphi'_c$. We first show that $\mathcal{T} \models \varphi'_c \rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$ (\clubsuit). Let (γ', ρ') be such that $(\gamma', \rho') \models \varphi'_c$. Any \mathcal{S} -derivable ML formula is satisfiable and hence there is $(\gamma, \rho) \models \varphi$. The hypothesis $\mathcal{T} \models \varphi \rightarrow (\exists \text{var}(\varphi_c))\varphi_c$ implies that there is ρ'' such that $(\gamma, \rho'') \models \varphi_c$. We may assume w.l.o.g. that $\text{var}(\varphi) \cap \text{var}(\varphi_c, \varphi'_c) = \emptyset$, which allows us to consider $\rho = \rho''$ and $\rho(x) = \rho'(x)$ for all $x \in \text{var}(\varphi'_c)$. It follows that $(\gamma', \rho) \models \varphi'_c$, which together with $(\gamma, \rho) \models \varphi \wedge \varphi_c$ implies $(\gamma', \rho) \models (\varphi \wedge \varphi_c)^{=?} \wedge \varphi'_c$. We obtain $(\gamma', \rho') \models (\exists \text{var}(\varphi_c, \varphi'_c))(\varphi \wedge \varphi_c)^{=?} \wedge \varphi'_c$. Since (γ', ρ') is arbitrary, it follows that the implication (\clubsuit) holds.

Since $\mathcal{T} \models \varphi'' \rightarrow \varphi'_c$ and $\mathcal{T} \models \varphi'_c \rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$ we obtain $\mathcal{T} \models \varphi'' \rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$ and hence [impl] can be applied for $\varphi'' \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$ as well. In this case the two homomorphic nodes have no children because [impl] is an axiom;

- the rule [der] is applied for $\varphi' \Rightarrow \varphi'_c$, hence φ' is \mathcal{S} -derivable. If φ'' is not satisfiable, then we can apply [impl]. Otherwise, since \mathcal{S} is total and $\mathcal{T} \models \varphi'' \rightarrow \varphi'$, it follows that φ'' is \mathcal{S} -derivable and hence [der] can be applied for $\varphi'' \Rightarrow \varphi'_c$ as well. The conclusion is obtained using the same reasoning to that done for the roots, but using Lemma B.1 instead of Lemma B.3;
- the rule [circ] is applied for $\varphi' \Rightarrow \varphi'_c$, hence there is $\varphi_d \Rightarrow \varphi'_d \in G$ such that $\mathcal{T} \models \varphi' \rightarrow (\exists \text{var}(\varphi_d))\varphi_d$. Since $\mathcal{T} \models \varphi'' \rightarrow \varphi'$, we obtain $\mathcal{T} \models \varphi'' \rightarrow (\exists \text{var}(\varphi_d))\varphi_d$ and hence [circ] can be applied for $\varphi'' \Rightarrow \varphi'_c$ as well. From $\mathcal{T} \models \varphi'' \rightarrow \varphi'$ we obtain $\mathcal{T} \models \Delta_{\varphi_d \Rightarrow \varphi'_d}(\varphi'') \rightarrow \Delta_{\varphi_d \Rightarrow \varphi'_d}(\varphi')$.

Now the proof of the lemma is finished. \square

We come back to the proof of the theorem. We show that the function f defined in the proof of Theorem 5.1 can be extended to the set of proof trees

$$PT(G) = \{P_{\varphi_c \Rightarrow \varphi'_c} \mid \varphi_c \Rightarrow \varphi'_c \in G\} \cup \{P_{\varphi \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)} \mid \varphi_c \Rightarrow \varphi'_c \in G, \mathcal{T} \models \varphi \rightarrow (\exists \text{var}(\varphi_c))\varphi_c\}$$

Since f has to "visit" a collection of proof trees, f has now three arguments: $f(\varphi, St, (\tau, \rho))$, where St is a list (stack) of pairs $\langle \varphi', P \rangle$ such that $P \in PT(G)$ and if $\langle \varphi', P \rangle$ is the first pair in St then $\varphi \Rightarrow \varphi'$ is the current node in P . The definition of f is as follows:

- (1) the current $\varphi \Rightarrow \varphi'$ in P corresponds to [impl], i.e. $\mathcal{T} \models \varphi \rightarrow \varphi'$:
 (a) P is the proof tree of the initial goal:

$$f(\varphi, \langle \varphi', T \rangle, (\tau, \rho)) = \varphi \quad (\text{B.2})$$

- (b) P is the proof tree of a circularity. In this case f goes back to the proof tree that used the circularity:

$$f(\varphi, \langle \varphi', P \rangle St, (\tau, \rho)) = f(\varphi, St, (\tau, \rho)) \quad (\text{B.3})$$

where St is not empty;

- (2) the current $\varphi \Rightarrow \varphi'$ in P corresponds to [der], i.e. φ is \mathcal{S} -derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^{\rho} hd(\tau')$. This case is similar to the proof of Theorem 5.1:

$$f(\varphi, St, (\tau, \rho)) = \varphi \Rightarrow_{\mathcal{S}}^{\rho} f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho)) \quad (\text{B.4})$$

- (3) the current $\varphi \Rightarrow \varphi'$ in P corresponds to [circ], i.e. $\mathcal{T} \models \varphi \rightarrow (\exists var(\varphi_c))\varphi_c$, $\varphi_c \Rightarrow \varphi'_c \in G$, and $\varphi' = \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)$. In this case f moves to the root of the proof tree of the circularity:

$$f(\varphi, St, (\tau, \rho)) = f(\varphi, \langle \varphi', P_{\varphi \Rightarrow \varphi'} \rangle St, (\tau, \rho)) \quad (\text{B.5})$$

The fact that function f is well-defined is ensured by the fact that the equations of the form $f(\dots) = f(\dots)$ cannot be consecutively applied infinitely times:

- the equation (B.3)) decreases the length of the finite list St ;
- the equation (B.5)) can be applied only once because the first rule used in $P_{\varphi \Rightarrow \varphi'}$ is [der].

Consequently, the rewriting relation \rightsquigarrow defined by (B.3)) and (B.5)), seen as rewrite rules, is terminating.

In the sequel we prove that the symbolic path computed by the function call $f(\varphi_c, \langle \varphi'_c, P_{\varphi_c \Rightarrow \varphi'_c} \rangle, (\tau, \rho))$ satisfies $\varphi_c \Rightarrow \varphi'_c$ and partially covers (τ, ρ) , for all $\varphi_c \Rightarrow \varphi'_c \in G$ and (τ, ρ) starting from φ_c . We assume that these symbolic paths are represented by expressions that are \rightsquigarrow -irreducible. We write $\varphi' \in St$ if St includes a pair $\langle \varphi', P \rangle$ and let $ST(\varphi \Rightarrow \varphi', (\tau, \rho))$ be the set of the lists St occurring in the proof tree of $f(nil, \varphi \Rightarrow \varphi', (\tau, \rho))$.

Lemma B.6 *The set $X = \{ \langle f(\varphi, St, (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \mid f(\varphi, St, (\tau, \rho)) \text{ a suffix of a symbolic path } f(\varphi_c, \langle \varphi'_c, P_{\varphi_c \Rightarrow \varphi'_c} \rangle, (\tau, \rho)) \text{ with } \varphi_c \Rightarrow \varphi'_c \in G, \varphi' \in St, \text{ and}$*

$\langle \tau, \rho \rangle$ a complete execution path starting from φ_c is backward closed w.r.t. to $(9, 10)$.

Proof Let $\langle f(\varphi, St, (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle$ be in X and assume that $St = \langle \varphi', T \rangle St'$, where St is a possibly empty list. We distinguish the following cases:

- $f(\varphi, \langle \varphi', P \rangle, (\tau, \rho)) = \varphi$ and $\mathcal{T} \models \varphi \rightarrow \varphi'$. We obtain $\langle f(\varphi, \langle \varphi', P \rangle, (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in (9, 10)(X)$ by (9).
- $f(\varphi, St, (\tau, \rho)) = \varphi \Rightarrow_S^s f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho))$, φ is \mathcal{S} -derivable, $\tau = \gamma_0 \Rightarrow_S \tau'$, and $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^p hd(\tau')$. Since $f(\varphi, \langle \varphi', T \rangle, (\tau, \rho))$ is a suffix of a symbolic path $f(\varphi_c, \langle \varphi'_c, T_{\varphi_c \Rightarrow \varphi'_c} \rangle, (\tau, \rho))$ with $\varphi_c \Rightarrow \varphi'_c \in G$ it follows that $f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho))$ is a suffix as well. We obtain $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho)), \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi' \rangle$ in X by further noticing that (τ', ρ) a complete execution path starting from $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$ by the definition of $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^p hd(\tau')$ and $(\gamma_0, \rho) \models \varphi \wedge \varphi_1$. Since $\mathcal{T} \models \varphi \rightarrow \varphi'$ we obtain $\langle f(\varphi, St, (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in (9, 10)(X)$ by (10).

□

Lemma B.7 *The set $Y = \{ \langle f(\varphi, St, (\tau, \rho)), (\tau, \rho) \rangle \mid f(\varphi, St, (\tau, \rho)) \text{ a suffix of a symbolic path } f(\varphi_c, \langle \varphi'_c, P_{\varphi_c \Rightarrow \varphi'_c} \rangle, (\tau, \rho)) \text{ with } \varphi_c \Rightarrow \varphi'_c \in G, \varphi' \in St, \text{ and } \langle \tau, \rho \rangle \text{ a complete execution path starting from } \varphi_c \}$ is backward closed w.r.t. to $(B.1, 8)$, where $(B.1, 8)$ is the set of rules given by (B.1) and (8).*

Proof Let $\langle f(\varphi, St, (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle$ be in X . We distinguish the following cases:

- $f(\varphi, \langle \varphi', P \rangle, (\tau, \rho)) = \varphi$ and $\mathcal{T} \models \varphi \rightarrow \varphi'$. We obtain $\langle f(\varphi, \langle \varphi', T \rangle, (\tau, \rho)), (\tau, \rho) \rangle \in (B.1, 8)(X)$ by (B.1).
- $f(\varphi, St, (\tau, \rho)) = \varphi \Rightarrow_S^s f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho))$, φ is \mathcal{S} -derivable, $\tau = \gamma_0 \Rightarrow_S \tau'$, and $\gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}}^p hd(\tau')$. The fact that $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho)), (\tau', \rho) \rangle$ is in Y is proved in a similar way to the proof of the membership $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho)), \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi' \rangle \in X$. From the hypothesis $\langle f(\varphi, St, (\tau, \rho)), (\tau, \rho) \rangle \in Y$ we obtain that (τ, ρ) starts from φ , i.e., $(\gamma_0, \rho) \models \varphi$. By using the rule (8) it follows that $\langle f(\varphi, St, (\tau, \rho)), (\tau, \rho) \rangle$ belongs to $(B.1, 8)(Y)$, which conclude this case.

In both cases we obtained that $\langle f(\varphi, St, (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in (B.1, 8)(X)$, which shows that X is backward closed. □

If $\varphi_c \Rightarrow \varphi'_c \in G$ then there is a proof three P_c of $\varphi_c \Rightarrow \varphi'_c$ under SCC. For each node $\Delta_{\varphi_d \Rightarrow \varphi'_d}(\varphi'' \Rightarrow \varphi'_c)$ in P_c corresponding to the [circ] inference rule, there is a proof tree P'' for $\varphi'' \Rightarrow \Delta_{\varphi_d \Rightarrow \varphi'_d}(\varphi'')$ under $\text{SCC}(G)$ by Lemma B.5. If (τ, ρ) is a complete execution path starting from φ , then $f(\varphi_c, \langle \varphi'_c, P_{\varphi_c \Rightarrow \varphi'_c} \rangle, (\tau, \rho))$ is defined and describes a symbolic path that par-

tially covers (τ, ρ) by Lemma B.7 and its corollary, and it satisfies $\varphi \Rightarrow \varphi'$ by Lemma B.6. By Lemma B.1 it follows that (τ, ρ) satisfies $\varphi \Rightarrow \varphi'$.

C Proofs of Results from Section 6

Proof of Lemma 6.1, Page 30. We first prove the lemma for $X, Y = \emptyset$ and then prove the general case.

(\subseteq) Consider an arbitrary $\gamma' \in \llbracket \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \rrbracket$, i.e., $(\gamma', \rho) \models \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$

for some valuation $\rho : Var \rightarrow \mathcal{T}$. We prove $\gamma' \in \llbracket \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi \rrbracket$, i.e., $(\gamma', \eta) \models \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi$ for some $\sigma \in \mu(\pi_1, \pi)$ and $\eta : Var \rightarrow \mathcal{T}$.

From $(\gamma', \rho) \models \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$ we obtain using Lemma 4.2 (which says that concrete execution steps backward-simulate symbolic ones) that there exists a configuration γ such that $(\gamma, \rho) \models \varphi \triangleq \pi \wedge \phi$ and $\gamma \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}} \gamma'$.

From $\gamma \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}} \gamma'$ we obtain a valuation ρ' such that $(\gamma, \rho') \models \varphi_1 \triangleq \pi_1 \wedge \phi_1$ and $(\gamma', \rho') \models \varphi_2 \triangleq \pi_2 \wedge \phi_2$. Using the hypothesis $var(\varphi) \cap var(\varphi_1, \varphi_2) = \emptyset$ we can choose ρ and ρ' such that $\rho = \rho'$.

We thus have $\gamma = \pi\rho = \pi_1\rho$, meaning that π_1 and π are A -unifiable (cf. Definition 6.1 and Remark 6.1). By Assumption 6.1 we obtain a substitution $\sigma : var(\pi_1) \rightarrow T_\Sigma(var(\pi)) \in \mu(\pi_1, \pi)$ such that $\pi \cong_A \pi_1\sigma$ and such that $\rho = \sigma\eta$ for some valuation $\eta : Var \rightarrow \mathcal{T}$. We extend σ to be the identity over $Var \setminus var(\pi_1)$, and then η can be chosen to coincide with ρ over $Var \setminus var(\pi_1)$. Thus, $x\rho \cong_A x\sigma\eta$ for all $x \in Var$. But since \cong_A is equality in the model \mathcal{T} , denoted by "=", we obtain $x\rho = x\sigma\eta$ for all $x \in Var$.

To conclude this direction of the proof we show $\gamma' \in \llbracket \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi \rrbracket$; specifically, we prove (\dagger) $(\gamma', \eta) \models \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi$:

- we have obtained $(\gamma', \rho) \models \varphi_2 \triangleq \pi_2 \wedge \phi_2$ above (since $\rho' = \rho$), hence, (i) $\gamma' = \pi_2\rho = \pi_2(\sigma\eta) = (\pi_2\sigma)\eta$, and $\rho = \sigma\eta \models \phi_2$, thus, (ii) $\eta \models \phi_2\sigma$;
- we have also obtained $(\gamma, \rho) \models \pi_1 \wedge \phi_1$, thus, $\rho = \sigma\eta \models \phi_1$ and then (iii) $\eta \models \phi_1\sigma$;
- moreover, we have obtained $(\gamma, \rho) \models \pi \wedge \phi$, thus, $\rho = \sigma\eta \models \phi$ and then $\eta \models \phi\sigma$. But σ is the identity everywhere except perhaps on $var(\pi_1)$ and using the hypothesis $var(\varphi) \cap var(\varphi_1, \varphi_2) = \emptyset$ we obtain $var(\phi) \cap var(\pi_1) = \emptyset$, thus, σ is the identity over $var(\phi)$ and from $\eta \models \phi\sigma$ we obtain (iv) $\eta \models \phi$.

From (i)-(iv) we obtain (\dagger) which concludes the proof of the \subseteq inclusion.

(\supseteq) Assume $\gamma' \in \llbracket \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi \rrbracket$, i.e., $(\gamma', \eta) \models \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi$ for some $\sigma \in \mu(\pi_1, \pi)$ and $\eta : \text{Var} \rightarrow \mathcal{T}$, and let $\rho \triangleq \sigma\eta$, where σ is the identity everywhere except perhaps on $\text{var}(\pi_1)$. We prove $(\gamma', \rho) \in \llbracket \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \rrbracket$.

For this, let $\gamma = \pi_1\rho$. We shall first prove (\clubsuit): $(\gamma, \rho) \models \varphi \triangleq \pi \wedge \phi$.

- We prove $\gamma = \pi\rho$: we have $\gamma = \pi_1\rho = \pi_1(\sigma\eta) = (\pi_1\sigma)\eta$ and since $\sigma \in \mu(\pi_1, \pi)$, $\pi_1\sigma \cong_A \pi$. Hence, $(\pi_1\sigma)\eta \cong_A \pi\eta$, and since \cong_A is the equality "=" in \mathcal{T} , we have $(\pi_1\sigma)\eta = \pi\eta$. Using the hypothesis $\text{var}(\varphi) \cap \text{var}(\varphi_1, \varphi_2) = \emptyset$ we obtain $\text{var}(\pi) \cap \text{var}(\pi_1) = \emptyset$, and since σ is the identity everywhere except perhaps on $\text{var}(\pi_1)$ and $\rho \triangleq \sigma\eta$ we obtain that η and ρ coincide on $\text{var}(\pi)$; hence, $\pi\eta = \pi\rho$. The above chain of equalities ensures $\gamma = \pi\rho$.
- We prove $\rho \models \phi$: from hypothesis $(\gamma', \eta) \models \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi$ we obtain in particular $\eta \models \phi$. Using again the hypothesis $\text{var}(\varphi) \cap \text{var}(\varphi_1, \varphi_2) = \emptyset$ we obtain that η and ρ coincide on $\text{var}(\phi)$, hence, $\rho \models \phi$ as well.

The statement (\clubsuit) is now proved. Next, we prove (\diamond): $(\gamma, \rho) \models \varphi_1 \triangleq \pi_1 \wedge \phi_1$. We have $\gamma = \pi_1\rho$ by the definition of γ . From $(\gamma', \eta) \models \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi$ we get $\eta \models \phi_1\sigma$ which implies $\sigma\eta \models \phi_1$, hence, $\rho = \sigma\eta \models \phi_1$, which proves (\diamond).

Finally, we prove (\heartsuit): $(\gamma', \rho) \models \varphi_2 \triangleq \pi_2 \wedge \phi_2$. We prove $\rho \models \phi_2$ by analogy with $\rho \models \phi_1$ above, and from $(\gamma', \eta) \models \pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi$ we get $\gamma' = (\pi_2\sigma)\eta = \pi_2(\sigma\eta) = \pi_2\rho$, which concludes the proof of (\heartsuit).

Recapitulating, we have (\clubsuit): $(\gamma, \rho) \models \varphi$, (\diamond): $(\gamma, \rho) \models \varphi_1$, and (\heartsuit) $(\gamma', \rho) \models \varphi_2$. From (\diamond) and (\heartsuit) we get $\gamma \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}} \gamma'$. Using that and (\clubsuit) and Lemma 4.1, which says that symbolic execution steps forward-simulate concrete ones, we obtain that there exists φ' such that $\varphi \Rightarrow_{\varphi_1 \Rightarrow \varphi_2}^s \varphi'$ and $(\gamma', \rho) \models \varphi'$. By definition of \Rightarrow^s , $\varphi' \triangleq \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$, which proves $(\gamma', \rho) \models \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$, i.e., $\gamma' \in \llbracket \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \rrbracket$ and the conclusion of the (\supseteq) inclusion follows.

This completes the proof of the lemma for $X, Y = \emptyset$.

For the general case: $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \triangleq (\exists \text{var}(\varphi_1, \varphi_2))(\varphi_1 \wedge (\exists X)\pi \wedge \phi)^{=?} \wedge \varphi_2$ is, due to the variable disjointness hypothesis in the lemma, ML-equivalent to the formula $(\exists X, \text{var}(\varphi_1, \varphi_2))(\varphi_1 \wedge \pi \wedge \phi)^{=?} \wedge \varphi_2$, i.e., to $(\exists X)\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\pi \wedge \phi)$.

To conclude the proof we apply the identity $\llbracket \varphi \rrbracket = \llbracket (\exists X)\varphi \rrbracket$, which holds for every ML formulas φ and set $X \subset \text{Var}$ of variables⁶, to the two members of the equality in the lemma's conclusion.

⁶ The equality is a simple consequence of Def. 3.8 of the ML satisfaction relation.

Proof of Theorem 6.1, Page 31. $\gamma \in \llbracket \Delta_{\mathcal{S}}(\varphi) \rrbracket$ iff there exists a rule $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$, with $\varphi_1 \triangleq \pi_1 \wedge \phi_1$, $\varphi_2 \triangleq (\exists Y)\pi_1 \wedge \phi_1$, such that $\gamma \in \llbracket \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \rrbracket$. Using Lemma 6.1 this happens if and only if $\gamma \in \llbracket (\exists X, Y)\pi_2 \sigma \wedge \phi_1 \sigma \wedge \phi_2 \sigma \wedge \phi \rrbracket$ for some $\sigma \in \mu(\pi_1, \pi)$.

By construction of the symbolic language, the rule $\alpha \triangleq \varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ induces a rule $\alpha^s \triangleq (\exists L)(\pi_1 \wedge^s \psi) \wedge \text{sat}(\phi_1 \wedge \psi) \Rightarrow (\exists L, Y)\pi_2 \wedge^s (\phi_2 \wedge \phi_1 \wedge \psi) \in \mathcal{S}^s$.

The rule α^s and substitution σ (N.B. a valuation in \mathcal{T}^s) generate the transition $\varphi \triangleq (\exists X)\pi \wedge^s \phi \Rightarrow_{\alpha^s} (\exists X, Y)\pi_2 \wedge^s (\phi_2 \sigma \wedge \phi_1 \sigma \wedge \phi) \triangleq \varphi'$. That is, φ interpreted as an element of $\mathcal{T}_{Cf_g^s}$ gives rise to a transition in \mathcal{S}^s to $\varphi' \in \mathcal{T}_{Cf_g^s}$, i.e., $\varphi \Rightarrow_{\mathcal{S}^s} \varphi'$.

Hence, $\gamma \in \llbracket \Delta_{\mathcal{S}}(\varphi) \rrbracket$ iff $\gamma \in \llbracket \varphi' \rrbracket$ (now interpreted as an ML formula in the signature of \mathcal{L}) for some transition $\varphi \Rightarrow_{\mathcal{S}^s} \varphi'$, which proves the theorem.