A Generic Framework for Symbolic Execution: Theory and Applications

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

A Generic Framework for Symbolic Execution: Theory and Applications

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

[a] *Faculty of Computer Science, "Alexandru Ioan Cuza" University of Iaşi, 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 language's syntax and execution infrastructure, a model interpreting the signature, and rewrite rules for the language's operational semantics. Then, symbolic execution amounts to 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 [26]. 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 [33]. The main advantage of symbolic execution is that it allows reasoning about multiple concrete

executions of a program, and its main disadvantage is the state-space explosion determined by decision statements and loops. Recently, the technique has found renewed interest in the formal-methods community due to new algorithmic developments and progress in decision procedures.

A symbolic program execution typically memorises symbolic values of program variables and a *path condition*, which accumulates constraints on the symbolic values on the path leading to the current instruction. When the next instruction to be executed is a conditional statement, whose condition depends on symbolic values, the execution is separated into distinct branches. The path condition is then updated to distinguish between the different branches. Then main foundational issues raised by symbolic execution include its relationships with the formal definition of the language, for soundness, and with the program logics, for applications to program analysis and verification.

Our contribution. The main contribution of the paper is a formal, language-independent theory and tool for symbolic execution, based on a language's operational semantics defined by term rewriting [1] . On the theoretical side, we define symbolic execution as the application of rewrite rules in the semantics by *derivation*, a logical description of symbolic successors of a given set of states also symbolically represented as a logical formula in Matching Logic (ML) [15]. 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) [15] 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 [38,42]. The proposed deductive system is proved to be sound using a coinductive proof technique. It is shown to be a strict generalization of an approach we presented in [28], 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 that the original 8-rule proof system given in [15]; the

[1] Most existing operational semantics styles (small-step, big-step, reduction with evaluation contexts, ...) have been shown to be representable in this way in [47].

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 [40], applied in this paper to programming language definitions (whereas in [40] it is applied to proving observational equalities between possibly infinite data structures, e.g., streams or trees). 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 [44]. This allows an uniform and rigorous approach for both finite and infinite symbolic executions.

On the practical side, we present an implementation of the theory into a prototype implementation based on in $\mathbb{K}$ [41], a framework dedicated to defining formal operational semantics of languages. Our current prototype is built on version 3.4 of $\mathbb{K}$, which 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 to 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 [39,42], which has beed used for verifying challenging C programs such as the Schorr-Waite garbage collector. MatchC also uses the formalism of Reachability Logic for program specifications; 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 [15], 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 [31] and verification techniques based on operational semantics [15,39].

Java PathFinder [34] 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 tools is fully dedicated to the Java language.

Another approach consists in combining concrete and symbolic execution into *concolic* execution. First, some concrete values given as input determine an execution path. When the program encounters a decision point, the paths not taken by concrete execution are explored symbolically. This has been implemented by several tools: DART [22], CUTE [46], EXE [10], PEX [17]. 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 [9], while others are more oriented towards verification [13,25,35]. Several techniques are implemented to improve the performance of these tools, such as *bounded verification* [11] and *pruning* the execution tree by eliminating redundant paths [16]. Other approaches offer support for verification of code contracts over programs. Spec# [7] is a tool developed at Microsoft that extends C# with constructs for non-null types, preconditions, postconditions, and object invariants. Spec# comes with a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks and multi-threading. A similar approach, which provides functionality for checking the correctness of a JAVA implementation with respect to a given UML/OCL specification, is the KeY [3] tool. In particular, KeY allows to prove that after running a method, its postcondition and the class invariant holds, using Dynamic Logic [23] and symbolic execution. The VeriFast tool [24] supports verification of single and multi-threaded C and Java programs annotated with preconditions and postconditions written in Separation Logic [36]. The Smallfoot tool [8,48] uses symbolic execution together with separation logic to prove Hoare triples. There are also approaches that attempt to automatically detect invariants in programs [32,45]. 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 performances, our generic and formal tool is, quite understandably, not in the same league as existing pragmatic tools, which are dedicated to specific languages and are focused on specific applications of symbolic execution. We focus here on language-independence: given a programming language defined in an algebraic/rewriting setting, we build its symbolic semantics and

use it for various analyses and verifications of programs in those languages.

Another body of related work is symbolic execution in term-rewriting systems. The technique called *narrowing*, initially used for solving equation systems in abstract datatypes, was extended for solving reachability problems in term-rewriting systems and was applied to the analysis of security protocols [30]. Such analyses rely on powerful unification-modulo-theories algorithms [19], 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 [5] presents our initial approach, and has since evolved into several directions. The present paper extends [5] 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 [5] 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 [6] expands another idea present in [5], 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 [6]. Moreover, the paper [43] 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 [6,43], neither of those papers deal with RL formula verification, nor do they employ (co)induction as a systematic, rig-

orous technique for reasoning about possibly infinite program executions. RL formula verification is dealt with in [28], but the procedure for RL verification proposed there is only a 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 (RL) 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 [29], 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-semantics/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, which we now explain with some examples.

A major feature of C++ expressions is the "sequenced before" relation [2], which defines a partial order over the evaluation of subexpressions. This can be easily expressed in $\mathbb{K}$ using the *strict* attribute to specify an evaluation order for an operation's operands. If the operator is annotated with the *strict* attribute then its operands will be evaluated in a nondeterministic order. For instance,

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

Fig. 1. Fragment of CinK syntax

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

Fig. 2. CinK configuration structure

all the binary operations are strict. Hence, they may induce non-determinism in programs because of possible side-effects in their arguments.

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

The $\mathbb{K}$ framework uses *configurations* to store program states. A configuration is a nested structure of cells, which typically include the program to be executed, input and output streams, values for program variables, and other additional information. The configuration of CinK (Figure 2) includes the $\langle\rangle_{\mathsf{k}}$ cell containing the code that remains to be executed, which is represented as a list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \ldots$ to be executed in the given order. Computation tasks are typically statements and expression evaluations. The memory is modelled using two cells $\langle\rangle_{\mathsf{env}}$ (which holds a map from variables to addresses) and $\langle\rangle_{\mathsf{store}}$ (which holds a map from addresses to values). The configuration also includes a cell for the function call stack $\langle\rangle_{\mathsf{stack}}$ and another one $\langle\rangle_{\mathsf{return}}$ for the return values of functions. The $\langle\rangle_{\mathsf{out}}$ cell holds the output of the program and is here connected to the standard output stream.

I_1 `+` I_2 $\Rightarrow$ $I_1 +_{Int} I_2$ [*plus*]

I_1 `/` $I_2 \wedge I_2 \neq_{Int} 0 \Rightarrow I_1 /_{Int} I_2$ [*division*]

`if(` *true* `)` St `else _` $\Rightarrow$ St [*if-true*]

`if(` *false* `) _ else` $St \Rightarrow St$ [*if-false*]

`while(` B `)` $St \Rightarrow$ `if(` B `){` St `while(` B `)` St `else {}}` [*while*]

V `;` $\Rightarrow \cdot$ [*instr-expr*]

$\langle$`++lval(` L `)` $\Rightarrow$ `lval(` L `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto (V \Rightarrow V +_{Int} 1) \cdots\rangle_{\mathsf{store}}$ [*inc*]

$\langle$`--lval(` L `)` $\Rightarrow$ `lval(` L `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto (V \Rightarrow V -_{Int} 1) \cdots\rangle_{\mathsf{store}}$ [*dec*]

$\langle\langle$`lval(` L `)`$= V \Rightarrow V \cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto \_ \Rightarrow V \cdots\rangle_{\mathsf{store}} \cdots\rangle_{\mathsf{cfg}}$ [*update*]

$\langle\langle$`$lookup(` L `)` $\Rightarrow V \cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto V \cdots\rangle_{\mathsf{store}} \cdots\rangle_{\mathsf{cfg}}$ [*lookup*]

`{` Sts `}` $\Rightarrow$ Sts [*block*]

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

When the configuration is initialised at runtime a CinK program is loaded in the $\langle\rangle_{\mathsf{k}}$ cell and all the other cells remain empty. A $\mathbb{K}$ *rule* is a topmost rewrite rule specifying transitions between configurations. Since usually only a small part of the configuration is changed by a rule, a *configuration abstraction* mechanism is used, allowing one to only specify the parts transformed by the rule. For instance, the (abstract) rule for addition, shown in Figure 3, represents the (concrete) rule

$$\begin{array}{l}\langle\langle I_1 \texttt{ + } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \\ \Rightarrow \\ \langle\langle I_1 +_{Int} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}\end{array} \quad (1)$$

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 rule for division has a side condition which restricts its application. The conditional statement `if` has two corresponding rules, one for each possible evaluation of the condition expression. The rule for the `while` loop performs an unrolling into an `if` statement and therefore it is not strict. Note that this unrolling does not generate infinite rewritings because the rule can be applied only when the statement `while` is on top of the cell `k`. The increment and update rules have side effects in the $\langle\rangle_{\mathsf{store}}$ cell, modifying the value stored at a specific address. Finally, the reading of a value from the memory is specified by the lookup rule, which matches a value in the $\langle\rangle_{\mathsf{store}}$ and places it in the $\langle\rangle_{\mathsf{k}}$ cell. The auxiliary construct `$lookup` is used when a program variable is evaluated as an rvalue.

In addition to these rules, written by the $\mathbb{K}$ user, the $\mathbb{K}$ framework automati-

cally generates so-called *heating* and *cooling* rules, which are induced by *strict* attributes, and ensure that the arguments declared to be strict are evaluated before the operation using those arguments. We show the case of division:

$$A_1 \ / \ A_2 \Rightarrow rvalue(A_1) \curvearrowright \square \ / \ A_2 \qquad rvalue(I_1) \curvearrowright \square \ / \ A_2 \Rightarrow I_1 \ / \ A_2$$
$$A_1 \ / \ A_2 \Rightarrow rvalue(A_2) \curvearrowright A_1 \ / \ \square \qquad rvalue(I_2) \curvearrowright A_1 \ / \ \square \Rightarrow A_1 \ / \ I_2$$

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 ([44] and [15], respectively) and Section 3.3 also reuses some of our own [5,28].

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 [44]. 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.

A *partially ordered set* is a pair $(L, \leq)$, where L is a non-empty set and $\leq$ is a reflexive, transitive and antisymmetric relation over L's elements. Given $X \subseteq L$, an element $x \in L$ is a *lower bound* for X if $x \leq y$, for all $y \in X$. Conversely, an element $y \in L$ is an *upper bound* for X if $x \leq y$, for all $x \in X$.

The *greatest lower bound* of X, denoted $glb(X)$, is a lower bound x such that $z \leq x$ for all lower bounds z of X. Similarly, the *least upper bound* of X, denoted $lub(X)$, is an upper bound y such that $y \leq z$ for all upper bounds z of X. We often write $lub(x, y)$ instead of $lub(\{x, y\})$ and $glb(x, y)$ instead of $glb(\{x, y\})$. Given a function $F : L \to L$, an element x 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.

A partially ordered set $(L, \leq)$ is a *lattice* if $lub(x, y)$ and $glb(x, y)$ exist for any $x, y \in L$. The lattice $(L, \leq)$ is *complete* if $lub(X)$ and $glb(X)$ exist for any $X \subseteq L$.

In the paper we also use the following notations:

- $x \vee y$ for $lub(x, y)$
- $x \wedge y$ for $glb(x, y)$
- $\top$ the greatest element of L
- $\bot$ the least element of L

Example 3.1 *Let S be a set. Then $(\mathcal{P}(S), \subseteq)$ is a complete lattice, where $\bot = \emptyset$, $\top = S$, $X \vee Y = X \cup Y$, $X \wedge Y = X \cap Y$.*

Theorem 3.1 (Knaster-Tarski) *Let $(L, \leq)$ be a complete lattice. Any monotone function $F : L \to L$ has a least fixed point $\mu Y.\, F(Y)$ (on short $\mu\, F$) and a greatest fixed point $\nu Y.\, F(Y)$ (on short $\nu\, F$).*

Moreover, $\mu\, F = \bigwedge\{X \mid F(X) \leq X\}$ and $\nu\, F = \bigvee\{X \mid X \leq F(X)\}$.

A set X is *backward closed* w.r.t. F if $X \leq F(X)$, and it is *forward closed* w.r.t. F if $F(X) \leq X$. A function $F : L \to L$ is *continuous* if $F(\bigvee_{n\geq 0} X_n) = \bigvee_{n\geq 0} F(X_n)$ for any increasing chain $X_0 \leq X_1 \leq \cdots$, and F is *cocontinuous* if $F(\bigwedge_{n\geq 0} X_n) = \bigwedge_{n\geq 0} F(X_n)$ for any decreasing chain $X_0 \geq X_1 \geq \cdots$.

Theorem 3.2 (Kleene) *If $F : L \to L$ is continuous then $\mu F = \bigvee_{n\geq 0} F^n(\bot)$. If $F : L \to L$ is cocontinuous then $\nu\, F = \bigwedge_{n\geq 0} F^n(\top)$.*

Definition 3.1 *Let U be a set. A* ground inference rule *on U is a pair (S, x), where $S \subseteq U$, $x \in U$. A set $\mathcal{R}$ of ground rules yields a function $\widehat{\mathcal{R}} : \mathcal{P}(U) \to \mathcal{P}(U)$ given by*

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

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

$$\frac{x_1, x_2, \ldots}{x}$$

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

It follows that each set of ground rules $\mathcal{R}$ satisfying the first hypothesis of the above proposition inductively defines a set $\mu\,\widehat{\mathcal{R}}$, and each set of ground rules $\mathcal{R}$ satisfying the second hypothesis of the above proposition coinductively defines a set $\nu\,\widehat{\mathcal{R}}$.

Example 3.2 *The system LIST, given below, coinductively defines the possibly infinite lists over integers:*

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

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

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

Definition 3.2 *Induction proof principle:*

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

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) $\subseteq X$.

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

Definition 3.3 *Coinduction proof principle:*

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

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).

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 3.4 (Proof Trees) *Let $\mathcal{R}$ be a set of ground rules over U and $x \in U$. 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 3.2 *Let $\mathcal{R}$ be a set of a set of ground rules over U such that $\mathcal{R}$ is cocontinuous. Then $x \in \nu\,\widehat{\mathcal{R}}$ iff there is a proof tree of x under $\mathcal{R}$.*

3.2 Matching Logic and Reachability Logic

This section is dedicated to presenting the logics that we will be using in the paper: matching logic and reachability logic. There are several versions of these recently introduced logics, we chose those presented in [15].

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

A *many-sorted signature* Σ consists of a set S of sorts and of a set of $S^* \times S$-sorted set of function symbols. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise infinite set of variables Var, let $T_\Sigma(Var)$ 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 $t_1, \ldots, t_n$ we let $var(t_1, \ldots, t_n) \triangleq var(t_1) \cup \cdots var(t_n)$. For any *substitution* $\sigma : Var \to T_\Sigma(Var)$ 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* $\rho : Var \to \mathcal{T}$ is extended to a (homonymous) Σ-algebra morphism $\rho : T_\Sigma(Var) \to \mathcal{T}$. The interpretation of a ground term t in $\mathcal{T}$ is denoted by $\mathcal{T}_t$. For simplicity, we often write $true, false, 0, 1, \ldots$ instead of $\mathcal{T}_{true}, \mathcal{T}_{false}, \mathcal{T}_0, \mathcal{T}_1, \ldots$ etc.

Definition 3.5 (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 $\mathcal{T}_p \subseteq \mathcal{T}_{s_1} \times \cdots \times \mathcal{T}_{s_n}$ for each $p \in \Pi_{s_1 \ldots s_n}$. Let Var denote a S-sorted set of*

variables. The set of (Σ, Π)-formulas *is defined by*

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

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

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

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

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

Matching Logic

We next recall the syntax and semantics of Matching Logic (ML) and Reachability Logic (RL) [15]. ML is a static logic of configurations, whereas RL is a dynamic logic of configurations, expressing their evolution over time. RL can be used both for specifying the operational semantics of programs (e.g., the rules in Fig. 3 denote RL formulas) and as a program-specification formalism.

Definition 3.6 (ML signature) *An ML signature is first-order signature* $\Phi = (\Sigma, \Pi, Cfg)$*, where* (Σ, Π) *is a many-sorted algebraic signature and* Cfg *is a distinguished sort in* Σ *for* configurations*. The set of* ML formulas *over* Φ *is defined by*

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

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 3.7 (Basic, Elementary, Quantified Elementary Patterns) *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

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

and the elementary pattern

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

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

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

Definition 3.8 (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 3.9 (ML Satisfaction) *The satisfaction relation $\models$ relates pairs (γ, ρ), where $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$, 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 $\rho' : Var \to \mathcal{T}$ such that $x\rho = x\rho'$ for all $x \in Var \setminus X$. If φ is an ML formula then $[\![\varphi]\!]$ denotes the set of concrete configurations $\{\gamma \mid (\gamma, \rho) \models \varphi$ for some $\rho\}$.*

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

Definition 3.10 (FOL encoding of ML [15]) *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 3.3 *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 3.3 ([15]) *Let φ be an ML formula and $\rho : Var \to \mathcal{T}$. Then $\rho \models \varphi^{=?}$ iff there is γ such that $(\gamma, \rho) \models \varphi$.*

Definition 3.11 *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 3.4 Consider the program `gcd` from Figure 4. The following RL formula

$$\langle\langle \texttt{gcd} \rangle_{\texttt{k}} \langle \texttt{a} \mapsto l_1 \ \texttt{b} \mapsto l_2 \rangle_{\mathsf{env}} \langle l_1 \mapsto a \ l_2 \mapsto b \rangle_{\mathsf{store}} \cdots \rangle_{\mathrm{cfg}} \wedge a \geq 0 \wedge b \geq 0 \Rightarrow$$
$$(\exists l, g) \langle \langle \cdot \rangle_{\texttt{k}} \langle (\texttt{x} \mapsto l) \ldots \rangle_{\mathsf{env}} \langle (l \mapsto g) \ldots \rangle_{\mathsf{store}} \cdots \rangle_{\mathrm{cfg}} \wedge g = gcd(a,b) \qquad (2)$$

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 $g = gcd(a, b)$ is stored. Here, $gcd()$ is a mathematical definition of the greatest-common-divisor operation (with $gcd(0, 0) = 0$ 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 propose a general notion of language definition based on algebraic specification and rewriting, the basics of which are assumed to be known to readers.

A language definition is a triple $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$, where

- Φ is an ML signature $(\Sigma, \Pi, \mathit{Cfg})$ 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 $\mathcal{T}_{\mathit{Cfg}}$ 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 $\langle\rangle_{\mathsf{k}}$ and $\langle\rangle_{\mathsf{env}}$ which contain the program to be executed, and respectively, a map from program variables to their values: $\langle\langle \mathit{Pgm}\rangle_{\mathsf{k}}\langle \mathit{Map}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. 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:

$$\langle X\ \cdots\rangle_{\mathsf{k}}\langle \cdots\ X \mapsto I\ \cdots\rangle_{\mathsf{env}} \Rightarrow \langle I\ \cdots\rangle_{\mathsf{k}}\langle \cdots\ X \mapsto I\ \cdots\rangle_{\mathsf{env}} \quad [\mathit{while\_lookup}]$$

$$\langle X = I\ ;\ \cdots\rangle_{\mathsf{k}}\langle \cdots\ X \mapsto \_\ \cdots\rangle_{\mathsf{env}} \Rightarrow \langle\ \cdots\rangle_{\mathsf{k}}\langle \cdots\ X \mapsto I\ \cdots\rangle_{\mathsf{env}} \quad [\mathit{while\_assign}]$$

$$\langle \texttt{while(}\ E\ \texttt{)}\ S\ \cdots\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}} \wedge \sigma[E] =_{\mathit{Bool}} \mathit{false} \Rightarrow \langle\ \cdots\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}} \quad [\mathit{while\_true}]$$

$$\langle \texttt{while(}\ E\ \texttt{)}\ S\ \cdots\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}} \wedge \sigma[E] =_{\mathit{Bool}} \mathit{true} \Rightarrow \langle S\ \texttt{while(}\ E\ \texttt{)}\ S\ \cdots\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}} \quad [\mathit{while\_false}]$$

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.

We now show how WHILE fits into our notion of language definition. The ML signature $(\Sigma, \Pi, \mathit{Cfg})$ 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., `_+_`: $\mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$, `_/_`: $\mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$, etc. The ML signature also contains a sort *Cfg* for configurations, which has a unique constructor $\mathit{Stmt} \times \mathit{Map} \to \mathit{Cfg}$ denoted by $\langle\ \langle\_\rangle_{\mathsf{k}}\ \langle\_\rangle_{\mathsf{env}}\ \rangle_{\mathsf{cfg}}$ (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 $\mathcal{T}_{\mathit{true}}$ 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 $\mathcal{T}_{\mathit{Cfg}}$. We give the relevant definitions using coinduction.

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

Definition 3.12 (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:

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

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

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

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 3.2); an execution path can be seen as a list of concrete configurations, where every two consecutive elements γ and γ' satisfy $\gamma \Rightarrow^{\rho}_{\mathcal{S}} \gamma'$. The largest set satisfied by these rules is the set of all finite and infinite execution paths.

Example 3.5 *Let* $\gamma \triangleq \langle\langle \mathtt{x} = 2; \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 3 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$ *and* $\gamma' \triangleq \langle\langle \cdot \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 2 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$ *be two concrete configurations, and* $\rho : Var \to \mathcal{T}$ *a valuation, such that* $X\rho = \mathtt{x}$ *and* $I\rho = 2$*, where X and I are variables from the assignment rule for* WHILE*. Then* $(\gamma \Rightarrow_{\mathcal{S}} \gamma', \rho)$ *is a (finite) execution path obtained using the rules from Definition 3.12. Execution paths can be infinite too, e.g.,* $\tau \triangleq \langle\langle \mathtt{while}(true)\, \mathtt{x}{=}1; \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 1 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \Rightarrow_{\mathcal{S}} \langle\langle \mathtt{x}{=}1; \mathtt{while}(true)\, \mathtt{x}{=}1; \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 1 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \Rightarrow_{\mathcal{S}} \langle\langle \mathtt{while}(true)\, \mathtt{x}{=}1; \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 1 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \Rightarrow_{\mathcal{S}} \cdots$*, which is obtained by applying the rules for* `while` *and assignment repeatedly.*

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

Definition 3.13 (Irreducible Configurations) *Let* $(\Phi, \mathcal{T}, \mathcal{S})$ *be a language definition. We say that* $\gamma \in \mathcal{T}_{Cfg}$ *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 3.14 (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:*

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

Example 3.6 *Recall* $\gamma' \triangleq \langle\langle \cdot \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 2 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$ *from the previous example. Then,* γ' *is irreducibile because there is no rule in the semantics of* WHILE *that can be applied to it. Another example of irreducible configuration is* $\gamma'' \triangleq \langle\langle 10/0 \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 2\ \mathtt{y} \mapsto 1 \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$*. 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' irreducibile 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 3.5 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 3.15 (Derivatives for ML and RL Formulas) *If φ is an ML formula then $\Delta_{\mathcal{S}}(\varphi) \triangleq \{(\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi)^{=?} \wedge \varphi_r \mid \varphi_l \Rightarrow \varphi_r \in \mathcal{S}\}$. 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 3.7 *Let $\varphi \triangleq \langle\langle X'{=}I'; \curvearrowright C'\rangle_{\mathsf{k}}\langle X' \mapsto V'\ M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ be an ML formula and $\varphi_l \Rightarrow \varphi_r$ be the RL formula corresponding to assignment in* WHILE*:*

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

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 assigment rule:

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

which simplifies to $\varphi' = \langle\langle C'\rangle_{\mathsf{k}}\langle X' \mapsto I'\ M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ using the fact that the equality $\langle\langle X = I; \curvearrowright C\rangle_{\mathsf{k}}\langle X \mapsto V\ M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} = \langle\langle X' = I'; \curvearrowright C'\rangle_{\mathsf{k}}\langle X' \mapsto V'\ M'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ 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 3.16 ($\mathcal{S}$-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 3.8 *Recall γ and γ' from Example 3.5, and φ and $\varphi_l \Rightarrow \varphi_r$ from Example 3.7. The ML formula φ is* $\mathcal{S}$-derivable *because $\varphi' \triangleq \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi)(\in \Delta_{\mathcal{S}}(\varphi))$ is satisfiable since $\gamma' \in [\![\varphi']\!]$.*

On the other hand, the ML formula $\widehat{\varphi} = \langle\langle I_1'/I_2'; \curvearrowright C'\rangle_{\mathsf{k}}\langle E'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2' =_{Int} 0$ *is not* $\mathcal{S}$*-derivabile. 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:*

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

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

$$(\exists I_1, I_2, C, E)\Big(\langle\langle I_1/I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} = \langle\langle I_1'/I_2'; \curvearrowright C'\rangle_{\mathsf{k}}\langle E'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0 \wedge I_2' =_{Int} 0\Big) \wedge \langle\langle I_1/_{Int} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

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

Definition 3.17 (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 3.1 *The semantics of CinK is not total because of the rules for division and modulo. The rule for division:* $\langle\langle I_1 \,/\, I_2 \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1/_{Int} I_2 \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}}$ *does not meet the condition of Definition 3.17 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* $\langle\langle I_1 \,/\, I_2 \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}} \wedge I_2 = 0 \Rightarrow \langle\langle \mathit{error} \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}}$ *that leads divisions by zero into "error" configurations. In this way, for any* γ *which is an instance of* $\langle\langle I_1 \,/\, I_2 \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}}$ *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 [15]. Again, we are using coinduction:

Definition 3.18 (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 writte* $(\tau, \rho) \models \varphi \Rightarrow \varphi'$ *if* $\langle(\tau, \rho), \varphi \Rightarrow \varphi'\rangle \in \nu\,\widehat{(3,4)}$*, where* $(3, 4)$ *is the system of ground rules:*

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

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

$\mathcal{S} \models \varphi \Rightarrow \varphi'$ *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 all* $\varphi \Rightarrow \varphi' \in F$. *We let* $[\![\varphi \Rightarrow \varphi']\!] \triangleq \{\tau \mid (\exists \rho)(\tau, \rho) \models \varphi \Rightarrow \varphi'\}$ *and* $[\![F]\!] = \bigcup_{\varphi \Rightarrow \varphi' \in F} [\![\varphi \Rightarrow \varphi']\!]$.

Example 3.9 *Recall the ML pattern* φ *and its derivative* φ' *from Example 3.7, and* γ *and* γ' *from Example 3.5. Let* $\rho : Var \to \mathcal{T}$ *be a valuation such that* $X\rho = X'\rho = \mathtt{x}$, $I\rho = I'\rho = 2$, $C\rho = C'\rho = \cdot$, *and* $V\rho = V'\rho = 3$. *Note that* $(\gamma', \rho) \models \varphi'$ *which implies, by the first rule in Definition 3.18, that* $(\gamma', \rho) \models \varphi' \Rightarrow \varphi'$ *(here,* γ' *is an execution path). Now, by the second rule in Definition 3.18, and the facts that* $(\gamma, \rho) \models \varphi_l$, $(\gamma, \rho) \models \varphi$, $\varphi_l \Rightarrow \varphi_r \in \mathcal{S}$, *and* $\varphi' = \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$ *we obtain* $(\gamma \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma', \rho) \models \varphi \Rightarrow \varphi'$.

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:*

$$\frac{}{\varphi} \ \varphi \text{ satisfiable} \tag{5}$$

$$\frac{\tau^{\mathfrak{s}}}{\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \tau^{\mathfrak{s}}} \ \varphi_0 \ \mathcal{S}\text{-derivable}, \ hd(\tau^{\mathfrak{s}}) \in \Delta_{\mathcal{S}}(\varphi_0) \tag{6}$$

where hd *is coinductively defined by*

$$hd(\varphi) = \varphi \qquad\qquad hd(\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \tau^{\mathfrak{s}}) = \varphi_0$$

Remark 4.1 *An alternative, equivalent definition for symbolic paths sconsists in requiring that* $hd(\tau^{\mathfrak{s}})$ *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 $\gamma_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma_2$ if there exists $\rho : \mathit{Var} \to \mathcal{T}$ such that $(\gamma_1, \rho) \models \varphi_l$ and $(\gamma_2, \rho) \models \varphi_r$, and $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\{\varphi_l \Rightarrow \varphi_r\}} \varphi_2$ 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:*

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

$$\frac{\langle \tau^{\mathfrak{s}}, (\tau, \rho) \rangle}{\langle \varphi_0 \Rightarrow^{\mathfrak{s}}_{\{\varphi_l \Rightarrow \varphi_r\}} \tau^{\mathfrak{s}}, (\gamma_0 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \tau, \rho) \rangle}\ (\gamma_0, \rho) \models \varphi_0 \tag{8}$$

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

Note that if $\tau^{\mathfrak{s}} \sqsupseteq (\tau, \rho)$ 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 $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\{\varphi_l \Rightarrow \varphi_r\}} \varphi_2$.*

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 $\tau^{\mathfrak{s}} \triangleq (\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \tau'^{\mathfrak{s}})$ such that $\tau^{\mathfrak{s}} \sqsupseteq (\tau, \rho)$.*

The precision property states that finite 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 $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\{\varphi_l \Rightarrow \varphi_r\}} \varphi_2$ 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 $\tau^{\mathfrak{s}}$, there exists (τ, ρ) such that $\tau^{\mathfrak{s}} \sqsupseteq (\tau, \rho)$.*

The theorems in this section 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 suceeds 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 $\tau^{\mathfrak{s}}$ be a symbolic execution path and $\varphi \Rightarrow \varphi'$ and RL formula. We say that $\tau^{\mathfrak{s}}$* satisfies *$\varphi \Rightarrow \varphi'$ and we write $\tau^{\mathfrak{s}} \models \varphi \Rightarrow \varphi'$ for $\langle \tau^{\mathfrak{s}}, \varphi \Rightarrow \varphi' \rangle \in \nu\, \widehat{(9,10)}$, where (9,10) is the system of the ground rules:*

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

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

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)

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

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

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

Specifically, if one can construct a finite proof tree under SYSTEP for a given RL formula then the formula belongs to $\nu\,\widehat{\mathsf{SYSTEP}}$. Theorem 5.1 then says that the formula holds in the semantics $\mathcal{S}$. An RL formula also holds if a 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

$$\langle\langle \texttt{while (x!=0) \{s=s+x; x=x-1;\}} \curvearrowright P\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto a\ \texttt{s} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow$$
$$(\exists b)\langle\langle P\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 0\ \texttt{s} \mapsto b\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge b =_{Int} \frac{a(a +_{Int} 1)}{2}$$

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 the infinitely many unfoldings 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.

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

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

T_2

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

$$\dfrac{}{\left\langle \begin{array}{l}\langle \texttt{s=s+x; x=x-1;while (x != 0) \{s=s+x; x=x-1;\}}\ P\rangle_{\mathsf{k}} \\ \langle \texttt{x} \mapsto a\ \texttt{s} \mapsto 0\rangle_{\mathsf{env}}\end{array}\right\rangle_{\mathsf{cfg}} \wedge a \neq_{Int} 0 \Rightarrow (\exists b)\langle\langle P\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 0\ \texttt{s} \mapsto b\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge b =_{Int} \dfrac{a(a+_{Int}1)}{2}}$$

T_1

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

$$\dfrac{T_1 \qquad T_2}{\langle\langle \texttt{while (x!=0) \{s=s+x; x=x-1;\}} \curvearrowright P\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto a\ \texttt{s} \mapsto 0\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow (\exists b)\langle\langle P\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 0\ \texttt{s} \mapsto b\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge b =_{Int} \dfrac{a(a+_{Int}1)}{2}}$$

Fig. 5. An infinite proof tree under SYSTEP

The following theorem, which we call *circularity principle*, states when the addition the circularity rule (and the circular reasoning that it allows) to SYSTEP does not compromise soundness. The main reason is to start not with G, but with $\Delta_{\mathcal{S}}(G)$, 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 $\varphi_c \Rightarrow \varphi'_c \in G$, $var(\varphi'_c) \subseteq var(\varphi_c)$. If $\Delta_{\mathcal{S}}(G) \subseteq \nu\,\widehat{\mathsf{SCC}(G)}$ then $\mathcal{S} \models G$.*

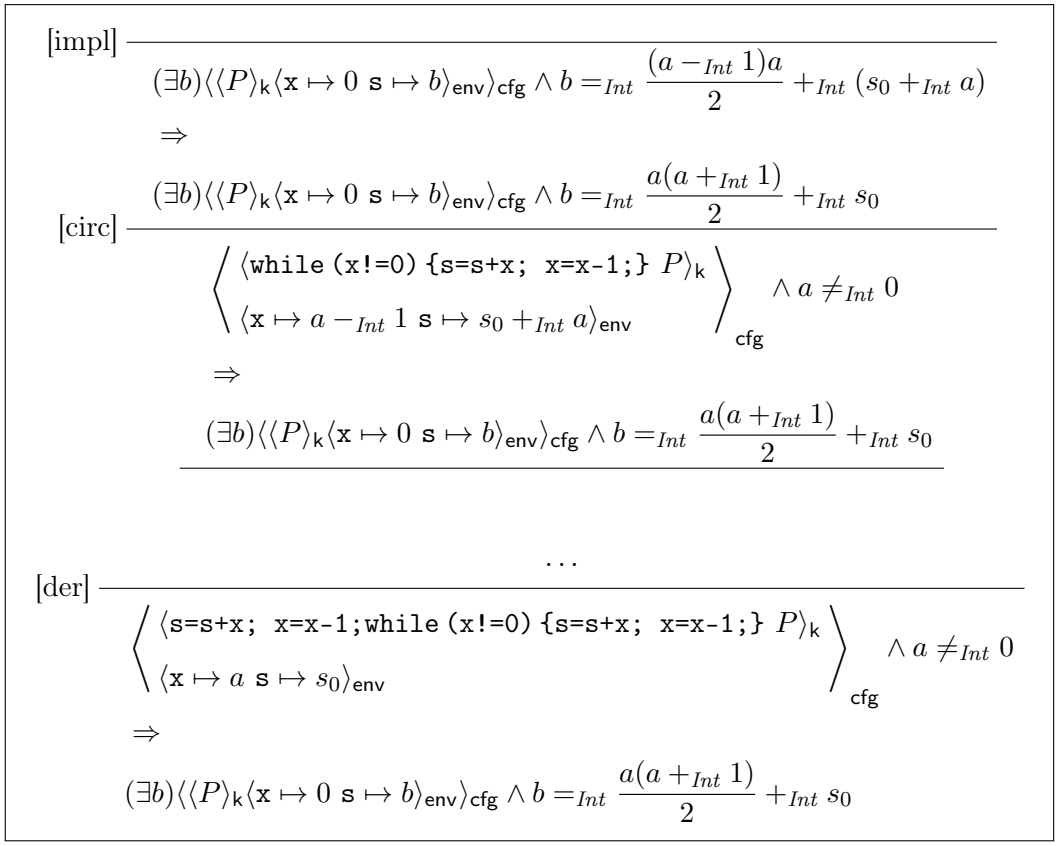

Fig. 6. The finite proof tree under SCC corresponding to the infinite proof tree T_2 in Fig. 5

Note that $\mathcal{S} \not\models \nu\widehat{\mathsf{SCC}(G)}$ in general. For instance, for any arbitrary set of RL formulas G, each $\varphi \Rightarrow \varphi' \in G$ is in $\nu\widehat{\mathsf{SCC}(G)}$ by applying the rule [circ]. Theorem 5.2 identifies a subset of proof trees under $\mathsf{SCC}(G)$ that are sound w.r.t. $\mathcal{S} \models \_$ (a proof tree for $\varphi \Rightarrow \varphi'$ under $\mathsf{SCC}(G)$ 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 $\mathsf{SCC}(G)$ is that it generates finite proof trees for a substantially larger set of RL formulas then SYSTEP. 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 SYSTEP 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 that the original 8-rule proof system for RL given in [15]: 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 [15], 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 [28]: that procedure can be seen a a strategy in SCC, in which implication is always applied before circularity, which is itself always applied before derivation.

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., $\Delta_{\mathcal{S}}(\varphi) \triangleq \{(\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi)^{=?} \wedge \varphi_r \mid \varphi_l \Rightarrow \varphi_r \in \mathcal{S}\}$ for ML formulas φ, and, respectively $\Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi') \triangleq \{\varphi_1 \Rightarrow \varphi' \mid \varphi_1 \in \Delta_{\mathcal{S}}(\varphi)\}$ 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 $\mathbb{K}$. This 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 $\Phi = (\Sigma, \Pi, \mathit{Cfg})$. 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., 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 $Tl_{\Sigma(\mathcal{D})\setminus\Sigma^{Data},Cfg}(Var)$, 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 will 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 $t' \in Tl_{\Sigma(\mathcal{D})\setminus\Sigma^{Data},Cfg}(Var)$ are* A-unifiable *if there is $\rho : Var \to \mathcal{T}$ 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 $t' \in Tl_{\Sigma(\mathcal{D})\setminus\Sigma^{Data},Cfg}(Var)$, we say that a substitution $\sigma : var(t') \to T_{\Sigma(\mathcal{D})}(Var^{Data})$ 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 $t' \in Tl_{\Sigma(\mathcal{D})\setminus\Sigma^{Data},Cfg}(Var)$, 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 $\rho' : Var \to \mathcal{T}$ 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 requirements [3] (i.e., the axioms are *linear, regular*, and *non-collapsing*). Combinations of the usual associativity, commutativity, and unity (ACU) axioms satisfy these requirements, and $\mathbb{K}$ language definitions intensively use these axioms (e.g., the k cell's content is an AU list, and, typically, 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 patters 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.

[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 [43] (Th. 3: Unification by Matching).

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* $\pi_1 \in Tl_{\Sigma(\mathcal{D})\setminus\Sigma^{Data},Cfg}(Var)$, *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,*

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

In a nutshell the 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 $\mathcal{L}^{\mathfrak{s}} = (\Phi^{\mathfrak{s}}, \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ over a *symbolic* ML signature $\Phi^{\mathfrak{s}} = (\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, Cfg^{\mathfrak{s}})$.

Symbolic Signature. The algebraic signature $\Sigma^{\mathfrak{s}}$ adds new sorts *Bool* and *Fol* to Σ and, for each predicate $p \in \Pi_{s_1,\ldots,s_n}$, a new operation $o_p : s_1 \times \cdots \times s_n \to Bool$. $\Sigma^{\mathfrak{s}}$ 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 $\_,\_ : IdList \times IdList \to IdList$. Finally, $\Sigma^{\mathfrak{s}}$ includes a sort $Cfg^{\mathfrak{s}}$ for symbolic configurations with its constructor $(\exists \_)\_ \wedge^{\mathfrak{s}} \_ : IdList \times Cfg \times Fol \to Cfg^{\mathfrak{s}}$. The set $\Pi^{\mathfrak{s}}$ consists of one single new predicate symbol: $sat : Fol \to Bool$.

Symbolic Model. The symbolic model $\mathcal{T}^{\mathfrak{s}}$ interprets syntactically all operations in $\Sigma^{\mathfrak{s}}$ except for the list concatenation operation, which is interpreted according to its (standard) equational definition. $\mathcal{T}^{\mathfrak{s}}$ interprets the single predicate symbol $sat : Fol \to Bool$ 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 $\mathcal{S}^{\mathfrak{s}}$ consists of the two following substeps:

- 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 $\pi_1 \in Tl_{\Sigma(\mathcal{D})\setminus\Sigma^{Data},Cfg}(Var)$, 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

$$(\exists L)(\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge sat(\phi_1 \wedge \psi) \Rightarrow (\exists L, Y)\pi_2 \wedge^{\mathfrak{s}} (\phi_2 \wedge \phi_1 \wedge \psi) \qquad (11)$$

  is created in $\mathcal{S}^{\mathfrak{s}}$, 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 $\mathcal{L}^{\mathfrak{s}}$ is now defined. Then we have a main result of this section, which says that derivatives can be computing by rewriting in $\mathcal{L}^{\mathfrak{s}}$.

Theorem 6.1 (Derivatives by Rewriting in $\mathcal{L}^{\mathfrak{s}}$) *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) occuring in $\mathcal{S}$. Then, $[\![\Delta_{\mathcal{S}}(\varphi)]\!] = \bigcup_{\varphi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \varphi'} [\![\varphi']\!]$.*

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 $\mathcal{L}^{\mathfrak{s}}$, 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 [5] 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

We have integrated our symbolic execution framework in version 3.4 of the $\mathbb{K}$ framework [41]. In $\mathbb{K}$, the definition of a language, say, $\mathcal{L}$, is compiled into a Maude [12] rewrite theory. Then, the $\mathbb{K}$ runner executes programs in $\mathcal{L}$ by applying the resulting rewrite rules to configurations containing programs. Our implementation follows the same process. The main difference is that our $\mathbb{K}$ compiler includes some additional transformations steps: rule linearisation, replacing the data subterms in left-hand-sides of rules with fresh variables and adding constraints to the rule's condition equating the fresh variables to the terms they replaced, adding a cell for path condition, and modifying semantical rules into symbolic rules of the form (11) as shown in the previous section. The effect is that the compiled rewrite theory we obtain defines a symbolic semantics of $\mathcal{L}$ instead of its concrete semantics. We note that the symbolic semantics can execute programs with concrete inputs as well. For user convenience we have also improved the $\mathbb{K}$ runtime environment with some specific options which are useful for providing programs with symbolic input and setting up an initial path condition. A conservative approximation of the predicate *sat* is implemented using the $\mathbb{K}$'s interface to the Z3 SMT solver [18].

Given a $\mathbb{K}$ language definition, our tool automatically generates its symbolic semantics. Thus, users that already have a $\mathbb{K}$ language definition can symbolically execute their programs without having to change anything in them. Actually, symbolic execution using the transformed definition amounts to applying the set of rules SYSTEP.

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 [4] 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 [4] since it uses conversions between arrays and pointers. We obtain the same result as [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>"
```

```
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`

```
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 [27] 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 $\mathcal{L}^{\mathfrak{s}}$ 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 $\mathsf{SCC}(G)$. 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 [27] 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 [14].

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

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

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

For the proof of `KMP` we use the original algorithm as presented in [14]. Another formal proof of the algorithm is given in [20] by using Why3 [21]. 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 [14]. We also prove that `KMP` finds *all* the occurrences of the pattern in the string, not only the first one. We let $P[1..i]$ denote the prefix of P of size i, and $P[i]$ denote its i-th element.

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

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

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

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

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

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

$$\Rightarrow$$

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

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

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

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

Loops can be annotated with invariants as shown below:

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

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

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

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

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

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 [14]. Second, we identified the properties listed above using a trial and error process, which also required manual 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.

8 Conclusion and Future Work

We have presented a formal and generic framework for the symbolic execution of programs in languages definable in an algebraic and term-rewriting setting. Symbolic execution is performed by applying a certain 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. These properties 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, which is now a part of the $\mathbb{K}$ framework, and its applications to the bounded model checking and deductive verification of nontrivial programs written in a subset of C++ also formally defined in $\mathbb{K}$.

Future Work We are planning to expand our tool, to make it able to seamlessly perform a wide range of program analyses, from testing and debugging to formal verifications, following ideas presented in related work, but with the added value of being language independent and grounded in formal methods. For this, we shall develop a rich domain of symbolic values, able to handle various kinds of data types. Formalising the interaction of symbolic-domain computations with symbolic execution is also a matter for future work.

Another future research direction is specifically targeted at our RL-formulas verifier, and aims at certifying its executions. The idea is to generate proof scripts for the Coq proof assistant [1], in order to obtain certificates that, despite any (inevitable) bugs in our tool, the proofs it generates are indeed correct. This amounts to, firstly, encoding our RL proof system in Coq, and proving its soundness with respect to the original proof system of RL (which have already been proved sound in Coq [39]). 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] The Coq proof assistant reference manual, `http://coq.inria.fr/refman/`.

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

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

[4] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. Model checking linear programs with arrays. *Electr. Notes Theor. Comput. Sci.*, 144(3):79–94, 2006.

[5] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A generic framework for symbolic execution. In *6th International Conference on Software Language*

Engineering, volume 8225 of *LNCS*, pages 281–301. Springer Verlag, 2013. Also available as a technical report at `http://hal.inria.fr/hal-00766220/`.

[6] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. Symbolic execution based on language transformation. *Computer Languages, Systems & Structures*, 2015. To appear, preprint available at `https://hal.inria.fr/hal-01186008`.

[7] 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.

[8] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Yi [48], pages 52–68.

[9] 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.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[11] 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.

[12] 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, July 2007.

[13] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Softw. Eng. Notes*, 26(5):142–151, 2001.

[14] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[15] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. 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 *LNCS*, pages 425–440. Springer, July 2014.

[16] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. *SIGPLAN Not.*, 48(4):329–342, March 2013.

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

[18] 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 *LNCS*, pages 337–340. Springer, 2008.

[19] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electr. Notes Theor. Comput. Sci.*, 238(3):103–119, 2009.

[20] Jean Christophe Filliâtre. Proof of kmp string searching algorithm. `http://toccata.lri.fr/gallery/kmp.en.html`.

[21] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[23] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.

[24] 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.

[25] 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.

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

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

[28] 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.

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

[30] 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.

[31] Brandon Moore and Grigore Roşu. Program verification by coinduction. Technical Report http://hdl.handle.net/2142/73177, University of Illinois, February 2015.

[32] Corina S. Păsăreanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *LNCS*, pages 164–181. Springer, 2004.

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

[34] Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors. *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM, 2010.

[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 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.

[39] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, 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.

[40] Grigore Roşu and Dorel Lucanu. Circular coinduction – a proof theoretical foundation. In *CALCO 2009*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.

[41] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

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

[43] Vlad Rusu, Dorel Lucanu, Traian-Florin Şerbănuţă, Andrei Arusoaie, Andrei Ştefănescu, and Grigore Roşu. Language Definitions as Rewrite Theories. *Journal of Logic and Algebraic Methods in Programming*, page 48, 2015. To appear, preprint available at `https://hal.inria.fr/hal-01186005`.

[44] 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/~sangio/DOC_public/corsoFL.pdf`.

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

[46] 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.

[47] Traian-Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

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

A Supplementary Background Material about Coinduction

The following definition is also inspired from [44]. We adapt it for our purpose, which is here to define functions on coinductively defined sets.

Definition A.1 (Game) *Let $\mathcal{R}$ be a set of a set of ground rules. A* game *in $\mathcal{R}$ consists of two players, V (the verifier) and R (the refuter), and an element x_0. V attempts to show that there is a proof tree for x under $\mathcal{R}$ and R attempts to show that there is no such a proof tree.*

During a play a data structure playingNodes*, storing the set of playing nodes, is maintained. A play consists of a sequence of alternative moves of the two players according to the following rules:*

- *the* play *starts with a move of V that chooses a rule $(S_0, x_0) \in \mathcal{R}$ (deriving x_0) and the set* playingNodes *is initialised with S_0;*
- *if it is the turn of R to move, then it chooses an element x_i from* playingNodes*;*
- *if it is the turn of V to move and the last element chosen by R is x_i, then x_i is removed from* playingNodes*, V chooses a rule $(S_i, x_i) \in \mathcal{R}$ (if any), and the nodes S_i are added to* playingNodes*;*

- *if* playingNodes *is empty then the play is over and V wins;*
- *if there is no a rule deriving the last element x_i chosen by R, then the play is over and R wins;*
- *if the play does not terminate, V wins.*

A play is described by a (finite or infinite) sequence

$$x_0, S_0, \ldots, x_n, S_n, \ldots$$

Since the two players are always the same, a game is denoted by the pair $(\mathcal{R}, x_0)$.

A strategy is a systematic way of moving for each player.

Definition A.2 (Strategy) *A* strategy for V *is a function that associates to each sequence*

$$x_0, S_0, \ldots, x_n$$

a rule $(S_n, x_n) \in \mathcal{R}$, whenever such a rule exists. A strategy for R *is a function that associates to each sequence*

$$x_0, S_0, \ldots, x_n, S_n,$$

an element from playingNodes $= S_0 \cup \cdots \cup S_n \setminus \{x_0, \ldots, x_n\}$*, whenever this set is not empty.*

A strategy is winning *for a player if it produces a win for that player in every play.*

In other words, a strategy for V says which rule is chosen when it is his turn to play, and a strategy for R says which playing node is chosen when it is he turn to play. We now use the above notions in order to define functions between coinductively defined sets, which we will use later in the paper. We note that the following definition does not appear in [44].

Definition A.3 (Functions over Coinductive Sets) *Let A and B two sets coinductively defined by $\mathcal{R}_A$ and $\mathcal{R}_B$, respectively. A coinductive function $f : A \to B$ is a function that maps each $x \in A$ and a given winning strategy for V in the game $(\mathcal{R}_A, x)$ into a winning strategy for V in the game $(\mathcal{R}_B, f(x))$.*

Example A.1 *Each list defined by the system LIST from the previous example has a unique winning strategy. A function $f : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$ that duplicate each element of a list, is specified by*

- *$f(nil) = nil$. The unique wining strategy α_{nil} for V in (LIST, nil) is that in which V chooses the rule A. The specification says that the strategy α_{nil} is mapped in itself.*

- $f(z\ \ell) = z * z\ f(\ell)$. *Let α be the wining strategy for V in* $(\mathrm{LIST}, z\ \ell)$. *This strategy chooses first the instance of the rule B corresponding to z and then it behaves like the wining strategy α' for V in* (LIST, ℓ). *The specification says that α is mapped into a strategy β that chooses first the instance of the rule B corresponding to $z * z$ and then behaves like the strategy associated to α'.*

B Proofs of Results from Section 4

Proof of Lemma 4.1, Page 21.
Let $\varphi_2 \triangleq \Delta_{\{\varphi_l \Rightarrow \varphi_r\}}(\varphi_1) \triangleq (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. Obviously $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\{\varphi_l \Rightarrow \varphi_r\}} \varphi_2$. We now prove $(\diamondsuit)\ (\gamma_2, \rho) \models (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. 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 $var(\varphi_1) \cap var(\varphi_l, \varphi_r) = \emptyset$, thus, we can construct a valuation ρ' such that $\rho'|_{var(\varphi_1)} = \rho$ and $\rho'|_{var(\varphi_l, \varphi_r)} = \eta$, and then $(\gamma_2, \rho') \models \varphi_r$ and $(\gamma_1, \rho') \models \varphi_l \wedge \varphi_1$. Using Proposition 3.3 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 $(\gamma_2, \rho) \models (\exists var(\varphi_l, \varphi_2))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$, which proves the lemma.

Proof of Theorem 4.1, Page 21. In this proof we use the following set of rules R:

$$\frac{}{\langle (\gamma, \rho), \varphi \rangle}\ (\gamma, \rho) \models \varphi \tag{B.1}$$

$$\frac{\langle (\tau, \rho), \varphi \rangle}{\langle (\gamma_0 \Rightarrow_{\mathcal{S}} \tau, \rho), \varphi_0 \rangle}\ (\gamma_0, \rho) \models \varphi_0, \varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi \tag{B.2}$$

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 $\nu\, \widehat{R}$:

- $sp(\langle (\gamma, \rho), \varphi \rangle) = \varphi$
- $sp(\langle (\gamma_0 \Rightarrow_{\mathcal{S}} \tau, \rho), \varphi_0 \rangle) = \varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} sp(\langle (\tau, \rho), \varphi \rangle)$.

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

We choose an arbitrary $\langle sp(\langle(\tau',\rho'),\varphi'\rangle),(\tau',\rho')\rangle \in Y$. Note that $\langle(\tau',\rho'),\varphi'\rangle \in \nu\,\widehat{R}$. On the one hand, if $\langle(\tau',\rho'),\varphi'\rangle$ was obtained using (B.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 (B.2) then there are γ', τ'', and φ'' such that $\tau' \triangleq \gamma' \Rightarrow_{\mathcal{S}} \tau''$, $(\gamma',\rho') \models \varphi'$, $\varphi' \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi''$, 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 $sp(\langle(\tau',\rho'),\varphi'\rangle) = \varphi' \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} sp(\langle(\tau'',\rho'),\varphi''\rangle)$, 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 (♣).

Second, we show that $X = \{((\tau,\rho),\varphi) | (hd(\tau),\rho) \models \varphi\}$ (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: $\{((\gamma,\rho),\varphi) \mid (\gamma,\rho) \models \varphi\} \cup \{((\gamma_0 \Rightarrow_{\mathcal{S}} \tau,\rho),\varphi_0) \mid (\gamma_0,\rho) \models \varphi, \varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi, (hd(\tau),\rho) \models \varphi\}$.

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 (B.1) we obtain $((\tau,\rho),\varphi) \in \widehat{R}(X)$.
- $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$. From $((\tau,\rho),\varphi) \in X$ we obtain $(hd(\tau),\rho) \models \varphi$, that is, $(\gamma_0,\rho) \models \varphi$. On the other hand, $\gamma_0 \Rightarrow_{\mathcal{S}} hd(\tau')$. By Lemma 4.1, there is φ' such that $(hd(\tau'),\rho) \models \varphi'$ and $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$. Since $(hd(\tau'),\rho) \models \varphi'$ then $((\tau',\rho),\varphi') \in X$. Thus, using (B.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 21.
From $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\{\varphi_l \Rightarrow \varphi_r\}} \varphi_2$ 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 $\rho'' : Var \to \mathcal{T}$ 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 3.3 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 $\rho''|_{var(\varphi_1)} = \rho|_{var(\varphi_1)}$, 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 (♠) $\gamma_1 \Rightarrow_{\{\varphi_l \Rightarrow \varphi_r\}} \gamma_2$. From $(\gamma_1,\rho'') \models$

$\varphi_l \wedge \varphi_1$ we obtain in particular $(\gamma_1, \rho'') \models \varphi_l$, and from $(\gamma_2, \rho'') \models (\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$ we obtain in particular $(\gamma_2, \rho'') \models \varphi_r$, which proves (♠) and the lemma.

Proof of Theorem 4.2, Page 22. We denote by T the set of all finite symbolic executions, which is the least fixpoint μR of the set of rules R from Definition 4.1.

Consider also the set of rules R'

$$\frac{}{\varphi}\ (\exists\gamma, \rho)(\gamma, \rho) \models \varphi, \tag{B.3}$$

$$\frac{\tau^{\mathfrak{s}}}{\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \tau^{\mathfrak{s}}}\ hd(\tau^{\mathfrak{s}}) \mathit{satisfiable}, (\forall\gamma', \rho)(\gamma', \rho) \models hd(\tau^{\mathfrak{s}}) \rightarrow (\exists\gamma)(\gamma, \rho) \models \varphi \wedge \gamma \Rightarrow_{\mathcal{S}} \gamma' \tag{B.4}$$

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

Next we prove (♠) that the set $Y = \{\tau^{\mathfrak{s}} \in \mu R | (\exists\tau, \rho)\tau^{\mathfrak{s}} \sqsupseteq (\tau, \rho)\}$ 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 $\tau'^{\mathfrak{s}} = \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \tau^{\mathfrak{s}}$ where $\tau^{\mathfrak{s}} \in Y$ (hence, $\tau^{\mathfrak{s}}$ covers some (τ, ρ)), $hd(\tau^{\mathfrak{s}})$ is satisfiable, and $(\forall\gamma')(\gamma', \rho) \models hd(\tau^{\mathfrak{s}}) \rightarrow (\exists\gamma)(\gamma, \rho) \models \varphi \wedge \gamma \Rightarrow_{\mathcal{S}} \gamma'$ holds (from the condition of the 2nd rule of R'). We have $(hd(\tau), \rho) \models hd(\tau^{\mathfrak{s}})$ from $\tau^{\mathfrak{s}} \sqsupseteq (\tau, \rho)$ thus, there exists γ such that $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_{\mathcal{S}} hd(\tau)$, which proves $\tau'^{\mathfrak{s}} = \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \tau^{\mathfrak{s}} \sqsupseteq (\gamma \Rightarrow_{\mathcal{S}} \tau, \rho)$ and thus the arbitrarily chosen $\tau'^{\mathfrak{s}} \in Y$ generated by the 2nd rule of R' is in Y.

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

C Proofs of Results from Section 5

We first introduce a definition and prove a lemma.

Definition C.1 *Consider the following rule:*

$$\frac{}{\langle \varphi, (\tau, \rho)\rangle}\ (hd(\tau), \rho) \models \varphi \tag{C.1}$$

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

If $\tau^{\mathfrak{s}} \gtrsim (\tau, \rho)$ then the concrete execution path can be longer than the symbolic one. We obviously have $\tau^{\mathfrak{s}} \sqsupseteq (\tau, \rho)$ (i.e., implies $\tau^{\mathfrak{s}} \gtrsim (\tau, \rho)$. We call $\gtrsim$ the *partially covers* relation, by contrast to the *covers* relation $\sqsupseteq$.

Lemma C.1 *For all* (τ, ρ) *and* $\varphi \Rightarrow \varphi'$*, if there exists* $\tau^{\mathfrak{s}}$ *such that* $\tau^{\mathfrak{s}} \models \varphi \Rightarrow \varphi'$ *and* $\tau^{\mathfrak{s}} \gtrsim (\tau, \rho)$ *then* $(\tau, \rho) \models \varphi \Rightarrow \varphi'$*.*

Proof We proceed by coinduction by showing that the set $X = \{\langle (\tau, \rho), \varphi \Rightarrow \varphi'\rangle \mid (\exists \tau^{\mathfrak{s}})\tau^{\mathfrak{s}} \gtrsim (\tau, \rho),\ \tau^{\mathfrak{s}} \models \varphi \Rightarrow \varphi'\}$ 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 $\tau^{\mathfrak{s}}$ which partially covers (τ, ρ) and $\tau^{\mathfrak{s}} \models \varphi \Rightarrow \varphi'$.

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

(1) $\tau^{\mathfrak{s}} = \varphi_0$. Since $\tau^{\mathfrak{s}} \models \varphi \Rightarrow \varphi'$ then $\mathcal{T} \models \varphi_0 \rightarrow \varphi \wedge \varphi'$ (cf. Definition 5.1). Also, from $\tau^{\mathfrak{s}} \gtrsim (\tau, \rho)$, that is, $\varphi_0 \gtrsim (\tau, \rho)$, we obtain $(hd(\tau), \rho) \models \varphi_0$ by (C.1), and thus, $(hd(\tau), \rho) \models \varphi \wedge \varphi'$ (cf. Definition 3.9). Hence $\langle (\tau, \rho), \varphi \Rightarrow \varphi'\rangle \in \widehat{(3, 4)}(X)$ by using (3).

(2) $\tau^{\mathfrak{s}} = \varphi_0 \Rightarrow^{\mathfrak{s}}_{\{\varphi_1 \Rightarrow \varphi_2\}} \tau'^{\mathfrak{s}}$, where $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$. From $\tau^{\mathfrak{s}} \models \varphi \Rightarrow \varphi'$ we obtain $\mathcal{T} \models \varphi_0 \rightarrow \varphi$ and $\tau'^{\mathfrak{s}} \models \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi \Rightarrow \varphi')$ (♠) (cf. Definition 3.18). On the other hand, from $\tau^{\mathfrak{s}} \gtrsim (\tau, \rho)$, we obtain $\tau = \gamma_0 \Rightarrow_{\{\varphi_1 \Rightarrow \varphi_2\}} \tau'$, $(\gamma_0, \rho) \models \varphi_0$, and $\tau'^{\mathfrak{s}} \gtrsim (\tau', \rho)$ (♢) (cf. Definition 4.2). Using (♠) and (♢) 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 $X \subseteq \nu\,\widehat{(3, 4)}$. □

Proof of Theorem 5.1, Page 23. We start by recalling the theorem:

If $\mathcal{S}$ *is total then* $\mathcal{S} \models \nu\,\widehat{\mathsf{SYSTEP}}$*.*

Let T be a proof tree of an RL formula under SYSTEP. We define a function $f(\varphi \Rightarrow \varphi', (\tau, \rho))$, for $\varphi \Rightarrow \varphi' \in T$ 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 $(hd(\tau), \rho) \models \varphi \wedge \varphi'$;
- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi', (\tau', \rho))$ if φ is $\mathcal{S}$-derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$.
  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 $\gamma_0 \Rightarrow^{\rho}_{\mathcal{S}} hd(\tau')$, then f chooses arbitrarily one of them. We have $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi' \in T$, 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 3.12 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 $X = \{\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \mid \varphi \Rightarrow \varphi' \in T, (\tau, \rho) \text{ starts from } \varphi\}$ 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 $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in \widehat{(9, 10)}(X)$.
- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi', (\tau', \rho))$, φ is $\mathcal{S}$-derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$. 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 $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi' \in T$ by the definition of proof trees under SYSTEP and (τ', ρ) is a complete execution path starting from $\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$ by the definition of $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$ 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 C.1 and prove our theorem. We prove that the set $Y = \{\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle \mid \varphi \Rightarrow \varphi' \in T, (\tau, \rho) \text{ starts from } \varphi\}$ is backwards closed w.r.t. $\widehat{(C.1, 8)}$. 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 $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \in \widehat{(C.1, 8)}(Y)$ by (C.1).
- $f(\varphi \Rightarrow \varphi', (\tau, \rho)) = \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi', (\tau', \rho))$, φ is $\mathcal{S}$-derivable,

$\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$. We prove that $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi', (\tau', \rho)) \in Y$ in a similar we proceeded for showing that $\langle f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi', (\tau', \rho)), \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi) \Rightarrow \varphi' \rangle \in X$. 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 $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle \in (\widehat{C.1, 8})(Y)$.

We have $Y \subseteq (\widehat{C.1, 8})(Y)$, and by the coinduction principle we obtain $Y \subseteq \nu\,(\widehat{C.1, 8})$, i.e. $f(\varphi \Rightarrow \varphi', (\tau, \rho)) \gtrsim (\tau, \rho)$ for each $\langle f(\varphi \Rightarrow \varphi', (\tau, \rho)), (\tau, \rho) \rangle \in Y$.

If $\varphi \Rightarrow \varphi' \in \nu\,\widehat{\mathsf{SYSTEP}}$ then there is a proof tree T of $\varphi \Rightarrow \varphi'$ under SYSTEP. 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 C.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 3.18.

Proof of Theorem 5.2, Page 24. 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 $\varphi_c \Rightarrow \varphi'_c \in G$, $var(\varphi'_c) \subseteq var(\varphi_c)$. If $\Delta_{\mathcal{S}}(G) \subseteq \nu\,\widehat{\mathsf{SCC}(G)}$ then $\mathcal{S} \models G$.

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

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

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

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

Proof We assume w.l.o.g. that $var(\varphi_1) \cap var(\varphi'_1, \varphi'_2) = \emptyset$ (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 $\varphi'_2 \triangleq (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi'_1)^{=?} \wedge \varphi_r$ (cf. Definition 3.15), 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 $\varphi_2 \triangleq (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. Obviously, $\varphi_2 \in \Delta_{\mathcal{S}}(\varphi_1)$. In order to show that $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_2$ 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:

$$\textit{For all } (\gamma_2, \rho), \textit{ if } (\gamma_2, \rho) \models \varphi'_2 \textit{ then } (\gamma_2, \rho) \models \varphi_2 \ (\spadesuit).$$

Let (γ_2, ρ) be an arbitrary pair such that $(\gamma_2, \rho) \models \varphi'_2$, that is, $(\gamma_2, \rho) \models (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi'_1)^{=?} \wedge \varphi_r$. By Definition 3.9, there is a valuation ρ' with $x\rho' = x\rho$ for all $x \notin var(\varphi_l, \varphi_r)$ 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 3.3, 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, $\mathcal{T} \models \varphi'_1 \rightarrow (\exists var(\varphi_1))\varphi_1$, which implies that $(\gamma_0, \rho') \models (\exists var(\varphi_1))\varphi_1$, wherefrom we obtain a valuation ρ'' such that $x\rho'' = x\rho'$ for all $x \notin var(\varphi_1)$ and $(\gamma_0, \rho'') \models \varphi_1$. Since $var(\varphi_1) \cap var(\varphi'_1, \varphi'_2) = \emptyset$ we can choose $x\rho'' = x\rho'$ for $x \in var(\varphi_1)$ too, 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 3.3) $\rho' \models (\varphi_l \wedge \varphi_1)^{=?}$. 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 $x \notin var(\varphi_l, \varphi_r)$ such that $(\gamma_2, \rho') \models (\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. By Definition 3.9 we obtain $(\gamma_2, \rho) \models (\exists var(\varphi_l, \varphi_r))(\varphi_l \wedge \varphi_1)^{=?} \wedge \varphi_r$. Hence, $(\gamma_2, \rho) \models \varphi_2$. Note that (γ_2, ρ) was arbitrarily chosen, and thus we have $\mathcal{T} \models \varphi'_2 \rightarrow \varphi_2$ ($\clubsuit$).

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

Thus, we conclude that $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_2$ is indeed a symbolic execution path.

There remains to prove $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_2 \sqsupseteq \varphi'_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'_2$. Let $\tau^{\mathfrak{s}} \triangleq \varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_2$ and $\tau'^{\mathfrak{s}} \triangleq \varphi'_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'_2$. We have to prove that for every execution path $\tau \triangleq \gamma_1 \Rightarrow_{\mathcal{S}} \gamma_2$, if $\tau'^{\mathfrak{s}} \sqsupseteq \tau$ then $\tau^{\mathfrak{s}} \sqsupseteq \tau$ (cf. Definition C.2). From $\tau'^{\mathfrak{s}} \sqsupseteq \tau$ 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 $\mathcal{T} \models \varphi'_1 \rightarrow (\exists var(\varphi_1))\varphi_1$ 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 $x \notin var(\varphi_1)$.

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

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

Corollary C.1 *Let $\varphi'_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'_2$ 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 $\varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_2 \sqsupseteq \varphi'_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'_2$ and $\mathcal{T} \models \varphi'_2 \rightarrow \varphi_2$.*

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

Lemma C.4 $G \subseteq \nu\,\widehat{\mathsf{SCC}(G)}$.

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

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

Proof We show that each rule applied in the construction of $T_{\varphi_c \Rightarrow \varphi'_c}$ (whose existence is known by Lemma C.4) can be used to construct T as well, provided that the left-hand side of the current node in T is satisfiable. Each time the same rule is applied for two corresponding nodes in T respectively $T_{\varphi_c \Rightarrow \varphi'_c}$, 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 T to $T_{\varphi_c \Rightarrow \varphi'_c}$ that maps the root of T into the root of $T_{\varphi_c \Rightarrow \varphi'_c}$ and each child of a node x in T 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 T the inference rule [impl] is applied and for its homomorphic image [der] is applied; in those cases the corresponding node in T has no children.

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

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

For the rest of the trees we distinguish the following cases (we assume that $\varphi' \Rightarrow \varphi'_c$ is the homomorphic image of $\varphi'' \Rightarrow \varphi'_c$ 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 $\mathcal{T} \models \varphi \rightarrow (\exists var(\varphi_c))\varphi_c$ implies that there is ρ'' such that $(\gamma, \rho'') \models \varphi_c$. We may assume w.l.o.g. that $var(\varphi) \cap var(\varphi_c, \varphi'_c) = \emptyset$, which allows us to consider $\rho = \rho''$ and $\rho(x) = \rho'(x)$ for all $x \in var(\varphi'_c)$. 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 $(\gamma', \rho') \models (\exists var(\varphi_c, \varphi'_c))(\varphi \wedge \varphi_c)^{=?} \wedge \varphi'_c$. 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 usign Lemma C.1 instead of Lemma C.3;
- the rule [circ] is applied for $\varphi' \Rightarrow \varphi'_c$, hence there is $\varphi_d \Rightarrow \varphi'_d \in G$ such that $\mathcal{T} \models \varphi' \rightarrow (\exists var(\varphi_d))\varphi_d$. Since $\mathcal{T} \models \varphi'' \rightarrow \varphi'$, we obtain $\mathcal{T} \models \varphi'' \rightarrow (\exists var(\varphi_d))\varphi_d$ 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. □

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

$$PT(G) = \bigcup_{\varphi_c \Rightarrow \varphi'_c \in G, \mathcal{T} \models \varphi \rightarrow (\exists var(\varphi_c))\varphi_c} T_{\varphi \Rightarrow \Delta_{\{\varphi_c \Rightarrow \varphi'_c\}}(\varphi)}.$$

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 $\langle \varphi', T \rangle$ such that $T \in PT(G)$ and if $\langle \varphi', T \rangle$ is the first pair in St then $\varphi \Rightarrow \varphi'$ is the current node in T. The

definition of f is as follows:

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

$$f(\varphi, \langle \varphi', T \rangle, (\tau, \rho)) = \varphi \tag{C.2}$$

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

$$f(\varphi, \langle \varphi', T \rangle \, St, (\tau, \rho)) = f(\varphi, St, (\tau, \rho)) \tag{C.3}$$

   where St is not empty;

(2) the current $\varphi \Rightarrow \varphi'$ in T corresponds to [der], i.e. φ is $\mathcal{S}$-derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$. This case is similar to the proof of Theorem 5.1:

$$f(\varphi, St, (\tau, \rho)) = \varphi \Rightarrow^{5}_{\mathcal{S}} f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau', \rho)) \tag{C.4}$$

(3) the current $\varphi \Rightarrow \varphi'$ in T 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:

$$f(\varphi, St, (\tau, \rho)) = f(\varphi, \langle \varphi', T_{\varphi \Rightarrow \varphi'} \rangle \, St, (\tau, \rho)) \tag{C.5}$$

The function f is well-defined, i.e. it defines a strategy for V in the game $((5, 6), f(x))$, where $(5, 6)$ is the set of ground rules defined by (5) and (6). This is ensured by the fact that the equations of the form $f(\ldots) = f(\ldots)$ cannot be consecutively applied infinitely times:

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

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

In the sequel we prove that the symbolic path computed by the function call $f(\varphi_c, \langle \varphi'_c, T_{\varphi_c \Rightarrow \varphi'_c} \rangle, (\tau, \rho))$ 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 reprezented by expressions that are $\rightsquigarrow$-irreducible. We write $\varphi' \in St$ id St includes a pair $\langle \varphi', T \rangle$ and $ST(\varphi \Rightarrow \varphi', (\tau, \rho))$ be the set of the lists St occurring in the proof tree of $f(nil, \varphi \Rightarrow \varphi', (\tau, \rho))$.

Lemma C.6 *The set* $X = \{\langle f(\varphi, St, (\tau, \rho)), \varphi \Rightarrow \varphi' \rangle \mid f(\varphi, St, (\tau, \rho))$ *a suffix of a symbolic path* $f(\varphi_c, \langle \varphi'_c, T_{\varphi_c \Rightarrow \varphi'_c} \rangle, (\tau, \rho))$ *with* $\varphi_c \Rightarrow \varphi'_c \in G$, $\varphi' \in St$, *and* (τ, ρ) *a complete execution path starting from* $\varphi_c\}$ *is backward closed w.r.t. to* $\widehat{(9, 10)}$.

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:

- $f(\varphi, \langle \varphi', T\rangle, (\tau,\rho)) = \varphi$ and $\mathcal{T} \models \varphi \rightarrow \varphi'$. We obtain $\langle f(\varphi, \langle \varphi', T\rangle, (\tau,\rho)), \varphi \Rightarrow \varphi'\rangle \in \widehat{(9,10)}(X)$ by (9).
- $f(\varphi, St, (\tau,\rho)) = \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau',\rho))$, φ is $\mathcal{S}$-derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$. 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 $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$ and $(\gamma_0,\rho) \models \varphi \wedge \varphi_1$. Since $\mathcal{T} \models \varphi \rightarrow \varphi$ we obtain $\langle f(\varphi, St, (\tau,\rho)), \varphi \Rightarrow \varphi'\rangle \in \widehat{(9,10)}(X)$ by (10).

□

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

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

- $f(\varphi, \langle \varphi', T\rangle, (\tau,\rho)) = \varphi$ and $\mathcal{T} \models \varphi \rightarrow \varphi'$. We obtain $\langle f(\varphi, \langle \varphi', T\rangle, (\tau,\rho)), (\tau,\rho)\rangle \in \widehat{(C.1, 8)}(X)$ by (C.1).
- $f(\varphi, St, (\tau,\rho)) = \varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} f(\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi), St, (\tau',\rho))$, φ is $\mathcal{S}$-derivable, $\tau = \gamma_0 \Rightarrow_{\mathcal{S}} \tau'$, and $\gamma_0 \Rightarrow^{\rho}_{\{\varphi_1 \Rightarrow \varphi_2\}} hd(\tau')$. 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 (8) it follows that $\langle f(\varphi, St, (\tau,\rho)), (\tau,\rho)\rangle \in \widehat{(C.1, 8)}(Y)$.

□

If $\varphi_c \Rightarrow \varphi'_c \in G$ then there is a proof three T_c of $\varphi_c \Rightarrow \varphi'_c$ under SCC. For each node $\Delta_{\varphi_d \Rightarrow \varphi'_d}(\varphi'' \Rightarrow \varphi'_c)$ in T_c corresponding to the $[\mathsf{circ}]$ inference rule, there is a proof tree T'' for $\varphi'' \Rightarrow \Delta_{\varphi_d \Rightarrow \varphi'_d}(\varphi'')$ under $\mathsf{SCC}(G)$ by Lemma C.5. If (τ,ρ) is a complete execution path starting from φ, then $f(\varphi_c, \langle \varphi'_c, T_{\varphi_c \Rightarrow \varphi'_c}, (\tau,\rho))$ is defined and describes a symbolic path that partially covers (τ,ρ) by Lemma C.7 and its corollary, and it satisfies $\varphi \Rightarrow \varphi'$ by Lemma C.6. By Lemma C.1 it follows that (τ,ρ) satisfies $\varphi \Rightarrow \varphi'$.

D Proofs of Results from Section 6

Proof of Lemma 6.1, Page 28. We first prove the lemma for $X, Y = \emptyset$ and then prove the general case.

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

for some valuation $\rho : \mathit{Var} \to \mathcal{T}$. We prove $\gamma' \in [\![\pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi]\!]$, 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 $\eta : \mathit{Var} \to \mathcal{T}$.

From $(\gamma', \rho) \models \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$ we obtain using Lemma 4.2 (which says that concrete execution steps backaward-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 $\mathit{var}(\varphi) \cap \mathit{var}(\varphi_1, \varphi_2) = \emptyset$ 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 $\sigma : \mathit{var}(\pi_1) \to T_\Sigma(\mathit{var}(\pi)) \in \mu(\pi_1, \pi)$ such that $\pi \cong_A \pi_1\sigma$ and such that $\rho = \sigma\eta$ for some valuation $\eta : \mathit{Var} \to \mathcal{T}$. We extend σ to be the identity over $\mathit{Var} \setminus \mathit{var}(\pi_1)$, and then η can be chosen to coincide with ρ over $\mathit{Var} \setminus \mathit{var}(\pi_1)$. Thus, $x\rho \cong_A x\sigma\eta$ for all $x \in \mathit{Var}$. But since $\cong_A$ is equality in the model $\mathcal{T}$, denoted by "=", we obtain $x\rho = x\sigma\eta$ for all $x \in \mathit{Var}$.

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

- 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 $\mathit{var}(\pi_1)$ and using the hypothesis $\mathit{var}(\varphi) \cap \mathit{var}(\varphi_1, \varphi_2) = \emptyset$ we obtain $\mathit{var}(\phi) \cap \mathit{var}(\pi_1) = \emptyset$, thus, σ is the identity over $\mathit{var}(\phi)$ and from $\eta \models \phi\sigma$ we obtain (iv) $\eta \models \phi$.

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

($\supseteq$) Assume $\gamma' \in [\![\pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi]\!]$, 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 $\eta : \mathit{Var} \to \mathcal{T}$, and let $\rho \triangleq \sigma\eta$, where σ is the identity everywhere except perhaps on $\mathit{var}(\pi_1)$. We prove $(\gamma', \rho) \in [\![\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)]\!]$.

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

- 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 $var(\varphi) \cap var(\varphi_1, \varphi_2) = \emptyset$ we obtain $var(\pi) \cap var(\pi_1) = \emptyset$, and since σ is the identity everywhere except perhaps on $var(\pi_1)$ and $\rho \triangleq \sigma\eta$ we obtain that η and ρ coincide on $var(\pi)$; 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 $var(\varphi) \cap var(\varphi_1, \varphi_2) = \emptyset$ we obtain that that η and ρ coincide on $var(\phi)$, hence, $\rho \models \phi$ as well.

The statement ($\clubsuit$) is now proved. Next, we prove ($\diamondsuit$): $(\gamma, \rho) \models \varphi_1 \triangleq \pi_1 \wedge \phi_1$. 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 ($\diamondsuit$).

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 ($\clubsuit$): $(\gamma, \rho) \models \varphi$, ($\diamondsuit$): $(\gamma, \rho) \models \varphi_1$, and ($\heartsuit$) $(\gamma', \rho) \models \varphi_2$. From ($\diamondsuit$) 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 $\varphi \Rightarrow^{\mathfrak{s}}_{\varphi_1 \Rightarrow \varphi_2\}} \varphi'$ and $(\gamma', \rho) \models \varphi'$. By definition of $\Rightarrow^{\mathfrak{s}}$, $\varphi' \triangleq \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$, which proves $(\gamma', \rho) \models \Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)$, i.e., $\gamma' \in [\![\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)]\!]$ and the conclusion of the ($\supseteq$) inclusion follows.

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

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

To conclude the proof we apply the identity $[\![\varphi]\!] = [\![(\exists X)\varphi]\!]$, which holds for every ML formulas φ and set $X \subset Var$ of variables [4], to the two members of the equality in the lemma's conclusion.

Proof of Theorem 6.1, Page 29. $\gamma \in [\![\Delta_{\mathcal{S}}(\varphi)]\!]$ iff there exists a rule $\varphi_1 \Rightarrow \varphi_2 \in S$, with $\varphi_1 \triangleq \pi_1 \wedge \phi_1$, $\varphi_2 \triangleq (\exists Y)\pi_1 \wedge \phi_1$, such that $\gamma \in [\![\Delta_{\{\varphi_1 \Rightarrow \varphi_2\}}(\varphi)]\!]$. Using Lemma 6.1 this happens if and only if $\gamma \in [\![(\exists X, Y)\pi_2\sigma \wedge \phi_1\sigma \wedge \phi_2\sigma \wedge \phi]\!]$ 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 $\alpha^{\mathfrak{s}} \triangleq (\exists L)(\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge sat(\phi_1 \wedge \psi) \Rightarrow (\exists L, Y)\pi_2 \wedge^{\mathfrak{s}} (\phi_2 \wedge \phi_1 \wedge \psi) \in \mathcal{S}^{\mathfrak{s}}$.

[4] The equality is a simple consequence of Def. 3.9 of the ML satisfaction relation.

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

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