Language-Independent Program Verification Using Symbolic Execution

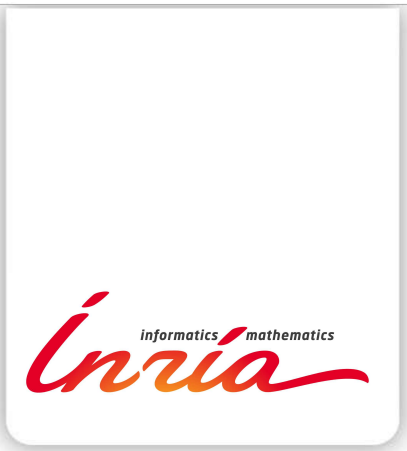
Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

Language-Independent Program Verification Using Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

RESEARCH REPORT
N° 8369
2013
Project-Team Dreampal

ISSN 0249-6399 ISRN INRIA/RR--8369--FR+ENG



Language-Independent Program Verification Using Symbolic Execution

Andrei Arusoaie*, Dorel Lucanu†, Vlad Rusu‡

Project-Team Dreampal

Research Report n° 8369 — 2013 — 28 pages

Abstract: We present an automatic, language-independent program verification approach and prototype tool based on symbolic execution. The program-specification formalism we consider is Reachability Logic, a language-independent alternative to Hoare logics. Reachability Logic has a sound and relatively complete deduction system that offers a lot of freedom to the user regarding the manner and order of rule application, but it lacks a strategy for automatic proof construction. Hence, we propose a procedure for proof construction, in which symbolic execution plays a major role. We prove that, under reasonable conditions on its inputs (the operational semantics of a programming language, and a specification of a program, both given as sets of Reachability Logic formulas) our procedure is partially correct: if it terminates it correctly answers (positively or negatively) to the question of whether the given program specification holds when executing the program according to the given semantics. Termination, of course, cannot be guaranteed, since program-verification is an undecidable problem; but it does happen if the provided set of goals includes enough information in order to be circularly provable (using each other as hypotheses). We introduce a prototype program-verification tool implementing our procedure in the $\mathbb{K}$ language-definition framework, and illustrate it by verifying nontrivial programs written in languages defined in $\mathbb{K}$.

Key-words: Program Verification Symbolic Execution, Language Independence.

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

RESEARCH CENTRE
LILLE – NORD EUROPE

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

Vérification de programmes indépendante des langages et basée sur l'exécution symbolique

Résumé : Nous présentons une méthode automatique pour vérifier des programmes, qui ne dépend pas du langage de programmation dans lequel les programmes à vérifier sont écrits. Pour celà nous nous appuyons sur la Reachability Logic, un formalisme de spécification introduit récemment, qui peut être vu comme une alternative à la logique de Hoare, mais qui, contrairement à cette dernière, ne dépend pas du langage de programmation utilisé. La Reachability Logic est munie d'un système déductif correct et relativement complet, qui laisse beaucoup de liberté à l'utilisateur sur la manière d'appliquer les règles de déduction, mais qui n'offre pas de stratégie pour construire automatiquement des preuves. Par conséquent nous proposons ici une procédure de construction des preuves, dans laquelle l'exécution symbolique joue un rôle essentiel. Nous montrons que, moyennant des conditions raisonnables sur la sémantique des langages de programmation et sur les propriétés des programmes, notre procédure est partiellement correcte. Ceci dit en substance que, lorsqu'elle termine, la procédure résout correctement le problème de vérification de programmes à base de Reachability Logic. La terminaison ne peut être garantie car la véritication de programmes est indécidable, mais la procéndure termine lorsque les propriétés contiennent suffisamment d'information pour être prouvées de manière circulaire, en s'utilisant mutuellement comme hypothèses. Nous présentons une implémentation prototype d'un outil de vérification basé sur ces idées, que nous avons implémenté dans la $\mathbb{K}$ framework et que nous illustrons sur des exemples de programmes non triviaux, écrits dans des langages formellement définis en $\mathbb{K}$.

Mots-clés : Vérification de programmes, Exécution symbolique, Indépendance aux langages.

1 Introduction

Reachability Logic (RL) [27] is a language-independent logic for specifying program properties. For instance, on the `gcd` program in Fig. 1, the RL formula

$$\langle\langle \texttt{gcd} \rangle_{\texttt{k}} \langle \texttt{a} \mapsto a \ \texttt{b} \mapsto b \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} \wedge a \geq 0 \wedge b \geq 0 \Rightarrow \exists M. \langle\langle \cdot \rangle_{\texttt{k}} \langle M \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} \wedge \mathit{lookup}(\texttt{x}, M) = \mathit{gcd}(a,b) \tag{1}$$

specifies that after the complete execution of the `gcd` program from a configuration where the program variables `a`, `b` are bound to non-negative values a, b, a configuration where the variable `x` is bound the value $\mathit{gcd}(a, b)$ is reached. Here, *gcd* is a mathematical definition of the greatest-

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

Figure 1: Program `gcd`

common-divisor ($\mathit{gcd}(0, 0) = 0$ by convention), and *lookup* is a standard lookup function in associative maps.

Reachability Logic can also be used for defining the operational semantics of programming languages, such as that of the language IMP in which the `gcd` program is written. A naive attempt at verifying the RL formula (1) consists in symbolically executing the semantics of the IMP language with the `gcd` progam in its left-hand side, i.e., running `gcd` with symbolic values $a, b \geq 0$ for `a`,`b`, and searching for a configuration matched by the formula's right-hand side. However, this does not succeed because the symbolic execution gets caught into the infinitely many iterations of the loop.

Independently of symbolic execution, the proof system of RL [27] is a set of seven inference rules that has been proved sound and relatively complete, meaning that, in principle, it proves all valid RL formulas (assuming an oracle that is able to decide the validity of first-order assertions). It is compact and elegant but, despite its nice theoretical properties, its use in practice on nontrivial programs is difficult, because it gives the user a lot of freedom regarding the order and manner of rule application, and offers no strategy for automatically constructing proofs. Moreover, it is not designed for disproving formulas: since the proof system is relatively complete, the only way to disprove a formula is to show that there exists no proof-tree for the formula, which is not practically possible since it requires exploring an infinite set of tentative proof trees.

Contribution. A language-independent procedure and prototype tool for program verification. The procedure uses symbolic execution as a main ingredient. It takes two inputs that are both sets of RL formulas: $\mathcal{S}$, the operational semantics of a programming language, and G, the specification of a program in the given language. The *program-verification problem* is: does the given program satisfy the specification G when executed according to the operational semantics $\mathcal{S}$? We show our procedure is *partially correct*, in the sense that, when it terminates, it correctly answers (either positively or negatively) to the question posed by the verification problem. On the one hand, our procedure does more that what the original proof system of RL (which may only answer positively when it terminates). On the other hand, unlike the original proof system, our procedure is not relatively complete, since it may not terminate for situations where the program does meet its specification, as illustrated by the naive symbolic-execution attempt at proving (1). In order to terminate our procedure typically requires additional user-provided

information under the form of additional RL formulas to be proved, such that the whole set of formulas can be proved in a conductive manner (using each other as hypotheses). The partial correctness of our procedure (decomposed into *soundness*, which, ensures that positive answers are correct, and *weak completeness*, which ensures correctness for negative answers) are based on certain mutual-simulation properties relating symbolic and concrete program execution, and on a so-called *circularity principle* for reachability-logic formulas, which specifies the conditions under which goals can be reused as hypotheses in proofs. This is essential for proving programs with infinite state-spaces induced e.g., by an unbounded (symbolic) number of loop iterations or of recursive calls. Soundness also requires that the semantics of the programming language is *total*; the behaviour of instructions is completely specified, and weak completeness moreover requires that the program specifications be precise (in a sense made formal in the paper) and that the semantics is *confluent* (in the usual sense of a confluent rewriting system) and *live* (i.e., it does not artificially force programs to terminate). We have implemented the approach as a prototype tool in the $\mathbb{K}$ framework [29]. We illustrate it on programs written in languages that are also defined in $\mathbb{K}$: a parallel program written in a parallel extension of the IMP language, and an implementation of the Knuth-Morris-Pratt string-searching algorithm [20], written in CINK [22], a formally specified subset of C++.

Organisation After this introduction, Section 2 presents preliminary concepts for the rest of the paper: a formal, generic framework for language definitions; the $\mathbb{K}$ language-definition framework as an instance of the proposed generic framework; an example of a simple imperative language defined in $\mathbb{K}$; and a brief presentation of Reachability Logic [27]. Section **??** contains the core contribution of the paper. We first present the main ingredients of a novel generic approach for symbolic execution. We then introduce our procedure for proof construction, whose properties (soundness, weak completeness) are based on a circularity principle for RL and on specific properties of symbolic execution (mutual simulation between symbolic and concrete executions) established in the previous section. Section 4 describes a prototype program-verification tool based on our language-independent symbolic execution tool [5] and its application to programs written in languages defined in $\mathbb{K}$. The paper ends with a description of related work. Two appendices contain, respectively, the proofs of the technical results in the paper, and a detailed description of applying our prototype tool on an example. The tool can be tried online on the examples in the paper (as well as other ones), at `https://fmse.info.uaic.ro/tools/kcheck`.

Acknowledgments This work was supported by the European grant POSDRU/159/1.5/S/137750 and partially by a BQR grant from the University of Lille.

2 Preliminaries

2.1 Language Definitions

We introduce generic language definitions in an algebraic and rewriting setting, whose main notions we assume to be known by readers. A language definition $\mathcal{L}$ is a triple $(\Phi, \mathcal{T}, \mathcal{S})$, where Φ is a many-sorted first-order signature, $\mathcal{T}$ is a Φ-model, and $\mathcal{S}$ is a set of semantical rules, described as follows.

Signature: Φ is a many-sorted first-order signature. It consists of a many-sorted algebraic signature Σ containing function symbols and of a set Π of predicate symbols. Σ includes at least a sort *Cfg* for *configurations* as well as sorts for the syntax of the language $\mathcal{L}$, e.g., expressions and statements. Σ may also include other data sorts, depending on the datatypes occuring in the language $\mathcal{L}$ (e.g., Booleans, integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all *data* sorts and their operations. We assume that the sort

Cfg and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{\mathsf{Data}}$. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise set of variables *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.

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

We use the *diagrammatic* notation for applying substitutions and valuations, i.e., a substitution/valuation is written after the term to which it is applied.

Rules: $\mathcal{S}$ is a set of semantical rules given as Reachability Logic (RL) formulas, defined below.

Definition 1 (pattern [27]) Patterns φ *over a set of variables Var are expressions defined by the the following grammar:*

$$\varphi ::= \pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid (\exists X)\, \varphi$$

where $\pi \in T_{Cfg}(Var), X \subseteq Var$. *An* elementary pattern *is a pattern of the form* $\pi \wedge \phi$, *where* $\pi \in T_{\Sigma,Cfg}(Var)$ *is a* basic pattern *and* ϕ *is a* Φ*-formula called the pattern's* condition. *The* satisfaction relation $(\gamma, \rho) \models \varphi$, *where* $\gamma \in \mathcal{T}_{Cfg}$ *and* $\rho : Var \to \mathcal{T}$, *is defined as follows:*
$(\gamma, \rho) \models \pi$, *where* π *is a basic pattern iff* $\pi\rho = e$,
$(\gamma, \rho) \models \neg\varphi'$ *iff* $(\gamma, \rho) \models \varphi'$ *does not hold,*
$(\gamma, \rho) \models \varphi_1 \wedge \varphi_2$ *iff* $(\gamma, \rho) \models \varphi_1$ *and* $(\gamma, \rho) \models \varphi_2$,
$(\gamma, \rho) \models (\exists X)\, \varphi'$ *iff* $\exists\ \rho' : Var \to \mathcal{T}$ *s.t.* $y\rho = y\rho'$ *for* $y \in Var \setminus X$ *and* $(\gamma, \rho') \models \varphi'$.
$[\![\varphi]\!]$ *denotes the set* $\{\gamma \in \mathcal{T}_{Cfg} \mid (\exists)\rho\colon Var \to \mathcal{T}.(\gamma, \rho) \models \varphi\}$. *We write* $\models \varphi$ *if* $(\gamma, \rho) \models$ *for all* γ, ρ.

Other first-order logical connectives (universal quantifiers, disjunction, implication, ...) may occur in patterns; they are defined from the basic connectives in the standard way. A basic pattern π defines a set of (concrete) configurations, and the condition ϕ gives additional constraints these configurations must satisfy. We identify basic patterns π with elementary patterns $\pi \wedge true$. Sample patterns are $\langle\langle I_1 \texttt{ + } I_2 \curvearrowright C\rangle_{\mathsf{k}} \langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \texttt{ / } I_2 \curvearrowright C\rangle_{\mathsf{k}} \langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$.

Definition 2 (RL formula [27]) *A RL formula (or* rule*) is a pair of patterns* $\varphi \Rightarrow \varphi'$.

Definition 3 (Transition System) *Any set* $\mathcal{S}$ *of rules defines a transition system* $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$ *such that* $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ *iff there exist* $\alpha \triangleq (\varphi \Rightarrow \varphi') \in \mathcal{S}$ *and* $\rho : Var \to \mathcal{T}$ *satisfying* $(\gamma, \rho) \models \varphi$ *and* $(\gamma', \rho) \models \varphi'$.

Remark 1 *Except when explicitly stated otherwise , in this paper we consider rules of the form* $\varphi \Rightarrow (\exists X)\varphi'$ *where* φ *and* φ' *are elementary patterns, and the variables in* $X \triangleq var(\varphi') \setminus var(\varphi)$ *are existentially quantified. Such rules generate the same transition system as the corresponding unquantified rules* $\varphi \Rightarrow \varphi'$, *thus, when we discuss transition systems generated by RL formulas we omit to write the existential quantifiers.*

2.2 A Simple Imperative Language and its Definition in $\mathbb{K}$

Our running example is IMP, a simple imperative language. The syntax of IMP is described in Figure 2 and is mostly self-explainatory since it uses a BNF notation. The statements of the language are either assignments, *if* statements, *while* loops, *nop* (i.e., the empty statement), blocks

$$
\begin{array}{l}
\mathit{Id} ::= \text{domain of identifiers} \qquad \mathit{Int} ::= \text{domain of integer numbers} \\
\mathit{Bool} ::= \text{domain of boolean constants} \\
\mathit{AExp} ::= \mathit{Int} \mid \mathit{Id} \mid (\mathit{AExp}) \mid \mathit{AExp} \texttt{ / } \mathit{AExp} \,[\text{strict}] \mid \mathit{AExp} \texttt{ * } \mathit{AExp} \,[\text{strict}] \\
\qquad \mid \mathit{AExp} \texttt{ + } \mathit{AExp} \,[\text{strict}] \mid \mathit{AExp} \texttt{ \% } \mathit{AExp} \,[\text{strict}] \\
\mathit{BExp} ::= \mathit{Bool} \mid (\mathit{BExp}) \mid \mathit{AExp} \texttt{ <= } \mathit{AExp} \,[\text{strict}] \\
\qquad \mid \texttt{not } \mathit{BExp} \,[\text{strict}] \mid \mathit{BExp} \texttt{ and } \mathit{BExp} \,[\text{strict(1)}] \\
\mathit{Stmt} ::= \texttt{\{\}} \mid \texttt{\{ } \mathit{Stmt} \texttt{ \}} \mid \mathit{Stmt} \texttt{ ; } \mathit{Stmt} \mid \mathit{Id} \texttt{ := } \mathit{AExp}, \,[\text{strict(2)}] \\
\qquad \mid \texttt{while } \mathit{BExp} \texttt{ do } \mathit{Stmt} \mid \texttt{if } \mathit{BExp} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \,[\text{strict(1)}] \\
\mathit{Code} ::= \mathit{AExp} \mid \mathit{BExp} \mid \mathit{Stmt} \mid \mathit{Code} \curvearrowright \mathit{Code}
\end{array}
$$

Figure 2: $\mathbb{K}$ Syntax of IMP

of statements, or sequential composition. The attribute *strict* in some production rules means that the arguments of the annotated expression/statement are evaluated before the expression/statement itself. If the attribute *strict* is followed by a list of numbers then it only concerns the arguments whose positions are in the list.

The operational semantics of IMP is given as rewrite rules over *configurations*. Configurations typically contain the program to be executed, together with any additional information required for program execution. The configuration structure depends on the language being defined; for IMP, it consists only of the program code to be executed and an environment mapping variables to values: $\mathit{Cfg} ::= \langle\langle \mathit{Code} \rangle_{\texttt{k}} \langle \mathit{Map}_{\mathit{Id},\mathit{Int}} \rangle_{\texttt{env}} \rangle_{\texttt{cfg}}$.

Configurations are written in $\mathbb{K}$ as nested structures of *cells*: for IMP, a top cell `cfg`, having a subcell `k` containing the code and a subcell `env` containing the environment. The code inside the `k` cell is represented as a list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \ldots$ to be executed in the given order. Computation tasks are typically statements and expressions. The environment in the `env` cell is a multiset of bindings of identifiers to values.

The semantics of IMP is shown in Figure 3. The rules say how configurations change when the first task from the `k` cell is executed. Dots in a cell mean that the rest of the cell remains unchanged. In addition to the rules in Fig. 3 the IMP semantics includes rules induced by *strict* attributes, which ensure that arguments of strict operators are pre-computed. For the `if` statement these are:

$$
\begin{array}{l}
\langle\langle \texttt{if } \mathit{BE} \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C \rangle_{\texttt{k}} \cdots \rangle_{\texttt{cfg}} \quad \Rightarrow \langle\langle \mathit{BE} \curvearrowright \texttt{if } \square \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C \rangle_{\texttt{k}} \cdots \rangle_{\texttt{cfg}} \\
\langle\langle B \curvearrowright \texttt{if } \square \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C \rangle_{\texttt{k}} \cdots \rangle_{\texttt{cfg}} \Rightarrow \langle\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \curvearrowright C \rangle_{\texttt{k}} \cdots \rangle_{\texttt{cfg}}
\end{array}
$$

Here $\square$ is a special variable, destined to receive the value of BE once it is computed, typically, by applying the other rules in the semantics.

IMP as a Language Definition. We show how the definition of IMP fits the theoretical framework given in Section 2.1. Nonterminals from the syntax (Int, Bool, $\mathit{AExp}, \ldots$) are sorts in Σ. Each production from the syntax defines an operation in Σ; e.g, the production $\mathit{AExp} ::= \mathit{AExp} \texttt{ + } \mathit{AExp}$ defines the operation $\_\texttt{+}\_ : \mathit{AExp} \times \mathit{AExp} \to \mathit{AExp}$. These operations define the constructors of the result sort. For the sort Cfg, the only constructor is $\langle\langle \_ \rangle_{\texttt{k}} \langle \_ \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} : \mathit{Code} \times \mathit{Map}_{\mathit{Id},\mathit{Int}} \to \mathit{Cfg}$. The expression $\langle\langle I_1 \texttt{ / } I_2 \curvearrowright C \rangle_{\texttt{k}} \langle \mathit{Env} \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} \wedge I_2 \neq_{\mathit{Int}} 0$ is an elementary pattern in which $=_{\mathit{Int}}$ is a predicate symbol, I_1, I_2 are variable of sort Int, C is a variable of sort Code (the rest of the computation), and Env is a variable of sort $\mathit{Map}_{\mathit{Id},\mathit{Int}}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets Int as the set of integers, the operations like $+_{\mathit{Int}}$ (cf. Figure 3) as the corresponding usual operation on integers, Bool as the set of Boolean values $\{\mathit{false}, \mathit{true}\}$, the operation like $\wedge$ as the usual Boolean operations, the sort $\mathit{Map}_{\mathit{Id},\mathit{Int}}$ as the

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

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

multiset of maps $X \mapsto I$, where X ranges over identifiers Id and I over the integers. Predicate symbols such as $=_{Int}, \leq_{Int}$ are interpreted by the corresponding predicates over integers. The value of an identifier X in an environment M is $\mathit{lookup}(X, M)$, and the environment M, updated by binding an identifier X to a value I, is $\mathit{update}(X, M, I)$. Here, $\mathit{lookup}()$ and $\mathit{update}()$ are operations in a signature $\Sigma^{\mathsf{Map}} \subseteq \Sigma^{\mathsf{Data}}$ of maps. The other sorts, *AExp*, *BExp*, *Stmt*, and *Code*, are interpreted in the algebra $\mathcal{T}$ as ground terms in which data subterms are replaced by their interpretations, e.g., `if` $1 >_{Int} 0$ `then {} else {}` is intepreted as `if` $\mathcal{D}_{true}$ `then {} else {}`.

2.3 Reachability Logic's Semantics and Proof System

We recall the semantics and proof system of Reachability Logic from [27]. These are essential for the correctness of our symbolic execution-based verification.

We assume a set $\mathcal{S}$ of RL formulas. A configuration γ is *terminating* if there is no infinite path in the transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$ starting in γ, and an RL formula $\varphi_1 \Rightarrow \varphi_2$ is *valid*, written $\mathcal{S} \models \varphi_1 \Rightarrow \varphi_2$, if for all terminating configurations γ_1 and valuations ρ satisfying $(\gamma_1, \rho) \models \varphi_1$, there is γ_2 such that $(\gamma_2, \rho) \models \varphi_2$ and $\gamma_1 \Rightarrow^*_{\mathcal{S}} \gamma_2$ in $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$. We consider here the version of the reachability logic proof system described in [27], reduced to the case of unconditional RL formulas (i.e., RL formulas whose application conditions may include first-order logic (FOL) formulas, but not other RL formulas). The proof system proves sequents of the form $\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'$ where G is a set of formulas called *circularities*. If $G = \emptyset$ then one simply writes $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$. A set of rules $\mathcal{S}$ is *weakly well-defined* if for each $\varphi \Rightarrow \varphi' \in S$ and for all valuations $\rho : \mathit{Var} \to \mathcal{T}$ there exists a configuration γ such that $(\gamma, \rho) \models \varphi'$. The deductive in Figure 4 is is *sound*: if $\mathcal{S}$ is weakly well-defined then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ implies $\mathcal{S} \models \varphi \Rightarrow \varphi'$. There is also a theoretical *relative completeness* result, which says that all valid formulas can be proved in the presence of an oracle deciding FOL formulas.

The proof system in Figure 4 leaves a lot of freedom to the user. For example, the Consequence rule allows the user to "invent" a new goal and to replace the current goal by the new

$$\text{[Axiom]}\ \frac{\varphi \Rightarrow \varphi' \in \mathcal{S} \quad \phi \quad \textit{FOL formula}}{\mathcal{S} \vdash_G \varphi \wedge \phi \Rightarrow \varphi' \wedge \phi}$$

$$\text{[Abstraction]}\ \frac{\mathcal{S} \vdash_G \varphi \Rightarrow \varphi' \quad X \cap \mathit{var}(\varphi') = \emptyset}{\mathcal{S} \vdash_G (\exists X.\, \varphi \Rightarrow \varphi')}$$

$$\text{[Reflexivity]}\ \frac{\cdot}{\mathcal{S} \vdash \varphi \Rightarrow \varphi}$$

$$\text{[Consequence]}\ \frac{\models \varphi_i \rightarrow \varphi_{i+1}, i \in \{1,3\} \quad \mathcal{S} \vdash_G \varphi_2 \Rightarrow \varphi_3}{\mathcal{S} \vdash_G \varphi_1 \Rightarrow \varphi_4}$$

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

$$\text{[Transitivity]}\ \frac{\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'' \quad (\mathcal{S} \cup G) \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'}$$

$$\text{[Circularity]}\ \frac{\mathcal{S} \vdash_{G \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'}$$

Figure 4: Proof System for RL.

one, provided some implication holds between the new and old goal's left and right-hand sides. The user is not guided in choosing such new goals. Most of the other rules allow such choices (the user must also choose which rule to apply next). There is currently no strategy for automatically constructing proofs. In the following sections we propose a procedure for automatic proof construction that, in addition to proving RL formulas is also able to disprove them.

3 Symbolic Execution for Reachability-Logic Verification

Symbolic execution consists in executing programs with symbolic values instead of concrete ones. A symbolic execution of a given sequence of instructions (or *path*) in a program evaluates the effect of the path on the program's configuration, which may typically contain symbolic variables in addition to constants. The effect of the path is encoded in a constraint on the symbolic variables called a *path condition*. We briefly present a novel approach to language-independent symbolic execution. We then show how symbolic execution can be used as a key ingredient for formal verification.

3.1 Symbolic Execution

Consider a language definition $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$. In order to symbolically execute programs in $\mathcal{L}$, one uses *symbolic configurations* consisting of Matching-Logic patterns of the form $\pi \wedge \phi$, with π a term with variables of sort *Cfg* and ϕ a Φ-formula denoting the path condition; and to apply the semantical rules of the language (possibly after some transformations preserving the semantics) to symbolic configurations instead of concrete ones. This generates the so-called symbolic transition relation:

Definition 4 (Symbolic transition relation) *Two elementary patterns φ, φ' are in the symbolic transition relation $\Rightarrow^{5}_{\mathcal{S}}$, denoted by $\varphi \Rightarrow^{5}_{\mathcal{S}} \varphi'$, iff $\varphi \triangleq \pi \wedge \phi$ and there exist: a rule $\alpha \triangleq$*

$\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ *with* $\varphi_i \triangleq \pi_i \wedge \phi_i$ *for* $i = 1, 2$, *and a substitution* σ *such that* $\pi_1\sigma = \pi$, *and* $\varphi' = \pi_2\sigma \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma$.

Example 1 *Consider the rule for division from the semantics of* IMP, *which we write in full form, which means replacing the ellipses by variables:* $\langle\langle I_1 \mathbin{/} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq 0 \Rightarrow \langle\langle I_1 /_{Int} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$. *Let* $\varphi \triangleq \langle\langle X \mathbin{/} Y \curvearrowright \cdot\rangle_{\mathsf{k}}\langle Y \mapsto A +_{Int} 1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A \neq_{Int} -1$. *The rule generates a symbolic transition from* φ' *to* $\langle\langle X \mathbin{/} Y \curvearrowright \cdot\rangle_{\mathsf{k}}\langle Y \mapsto A +_{Int} 1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A \neq_{Int} -1 \wedge Y \neq_{Int} 0$.

Remark 2 *The symbolic transition relation* $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ *is finitely branching if* $\mathcal{S}$ *is a finite set of rules.*

The two following theorems ensure that the symbolic and concrete transition systems mutually simulate each other; they are the properties one naturally expects from symbolic execution, ensuring that analyses based ob symbolic execution carry over to concrete program executions. They will be used for proving the correctness of our program-verification procedure. They rely on the following:

Assumption 1 *For each rule* $\alpha \triangleq \varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$:

- *the left-hand side* $\varphi_1 \triangleq \pi_1 \wedge \phi_1$ *is linear and all its sub terms of data sorts are variables;*
- $var(\varphi_2) \setminus var(\varphi_1)$ *contains at most variables of data sort.*

Remark 3 *The first item of the above assumption can always be satisfied after some transformation of the left-hand side pattern* $\pi_1 \wedge \phi_1$ *into another pattern* $\pi'_1 \wedge \phi'_1$ *satisfying* $[\![\pi_1 \wedge \phi_1]\!] = [\![\pi'_1 \wedge \phi'_1]\!]$, *and such that* π'_1 *is linear and all its data subterms are variables. For this, just replace all duplicated variables and all non-variable data subterms in* π_1 *by fresh variables, and add constraints to equate in* ϕ_1 *these variables to the subterms they replaced.*

Example 2 *The basic pattern* $\langle\langle \texttt{if}\ true\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$ *has the non-variable term* $true$. *It is thus replaced by the following pattern:* $\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge B =_{Bool} true$.

Theorem 1 (Coverage) *For every concrete execution* $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ *and pattern* φ_0 *over variables of data sorts such that* $\gamma_0 \in [\![\varphi_0]\!]$, *there is a symbolic execution* $\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_n \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ *such that* φ_i *only has variables of data sorts and* $\gamma_i \in [\![\varphi_i]\!]$ *for* $i = 0, 1, \ldots$.

Theorem 2 (Precision) *For every symbolic execution* $\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_1 \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi_n$ *and every* $\gamma_n \in [\![\varphi_n]\!]$ *there is a concrete execution* $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n$ *such that* $\gamma_i \in [\![\varphi_i]\!]$ *for* $i = 0, \ldots n$.

3.2 A Procedure for RL-based Program Verification

We introduce in this section our procedure for program verification. We first define, using the symbolic transition relation in Definition 4, the *derivative* operation used in our procedure.

Definition 5 (Derivative) *The* derivative $\Delta_{\mathcal{S}}(\varphi)$ *of an elementary pattern* φ *for a set* $\mathcal{S}$ *of rules is* $\Delta_{\mathcal{S}}(\varphi) \triangleq \bigvee_{\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'} \varphi'$. *We say that* φ *is* derivable for $\mathcal{S}$ *if* $\Delta_{\mathcal{S}}(\varphi)$ *is a nonempty disjunction.*

Remark 4 *Since the symbolic transition relation is finitely branching (Remark 2), for finite rule sets* $\mathcal{S}$ *the derivative is a finite disjunction.*

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

Definition 6 (Cover) *Consider an elementary pattern $\varphi \triangleq \pi \wedge \phi$. A set of rules $\mathcal{S}'$ such that for each $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'$ there exists a substitution $\sigma_\pi^{\pi_1}$ such that $\pi_1 \sigma_\pi^{\pi_1} = \pi$, and satisfying $\models \phi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} (\phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, is a* cover *of φ.*

Remark 5 *$\sigma_\pi^{\pi_1}$ in Definition 6 is a syntactical matcher of π_1 onto π, hence it is unique if it exists.*

The notion of *total* semantics is also essential for the soundness of our approach.

Definition 7 *$\mathcal{S}$ is total if, for all patterns φ derivable for $\mathcal{S}$, $\mathcal{S}$ is a cover for φ.*

Remark 6 *The semantics of* IMP *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 7 because the semantics of* IMP *does not cover patterns of the form $\langle\langle I_1 \,/\, I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}}$, which are derivable (by the division rule). This is due to the condition $I_2 \neq 0$, which is not logically valid. The semantics can easily be made total by adding a rule $\langle\langle I_1 \,/\, I_2 \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}} \wedge I_2 = 0 \Rightarrow \langle\langle error \;\cdots\rangle_{\mathsf{k}} \;\cdots\rangle_{\mathsf{cfg}}$ that leads divisions by zero into "error" configurations. We assume hereafter that the* IMP *semantics has been transformed into a total one by adding the above rule.*

We now have almost all the ingredients for proving RL formulas by symbolic execution. Assume a language with a semantics $\mathcal{S}$, and a finite of RL formulas with elementary patterns in their left-hand sides $G = \{\varphi_i \Rightarrow \varphi_i' \mid i = 1, \ldots, n\}$. We say that a RL formula $\varphi \Rightarrow \varphi'$ is *derivable* for $\mathcal{S}$ if the left-hand side φ is derivable for $\mathcal{S}$. If G is a set of RL formulas then $\Delta_{\mathcal{S}}(G)$ is the set $\{\Delta_{\mathcal{S}}(\varphi) \Rightarrow \varphi' \mid \varphi \Rightarrow \varphi' \in G\}$, and $\mathcal{S} \models G$ denotes $\bigwedge_{\varphi \Rightarrow \varphi' \in G} \mathcal{S} \models \varphi \Rightarrow \varphi'$. The following theorem is essential for the soundness of our verification procedure. It is a *circularity principle* also encountered in other coinductive frameworks, e.g., [28].

Theorem 3 (Circularity Principle for RL) *If $\mathcal{S}$ is total and G is derivable for $\mathcal{S}$, then $\mathcal{S} \cup G \vdash \Delta_{\mathcal{S}}(G)$ implies $\mathcal{S} \models G$.*

A Procedure for Systematic Proof Construction. There remains to define our procedure for proof construction and to establish its partial correctness (i.e., soundness and weak completeness). Given a RL formula $\varphi \Rightarrow \varphi'$ such that $\varphi = \bigvee_{i \in I} \varphi_i$ and φ_i is an elementary pattern for all $i \in I$, we denote by $split(\varphi \Rightarrow \varphi')$ the set of formulas $\{\varphi_i \Rightarrow \varphi' \mid i \in I\}$. Given a set of RL formulas G, $split(G) \triangleq \bigcup_{\varphi \Rightarrow \varphi'} split(\varphi \Rightarrow \varphi')$. We say a rule $\alpha \triangleq \pi_\alpha \wedge \phi_\alpha \Rightarrow \varphi'_\alpha$ is *applicable* to a formula $\varphi \Rightarrow \varphi'$ if α covers φ and φ'_α is derivable for $\mathcal{S}$ iff φ' is derivable for $\mathcal{S}$. Our procedure is shown in Figure 5.

Theorem 4 (Soundness) *Assume that for a total semantics $\mathcal{S}$ and a set of goals G_0 derivable for $\mathcal{S}$, the call $prove(S \cup G_0, split(\Delta_{\mathcal{S}}(G_0)))$ returns true. Then $\mathcal{S} \models G_0$.*

Example 3 *We show how the RL formula (1) is proved using our procedure, which amounts to verifying that the* `gcd` *program meets its specification. For this, we consider the following auxiliary formula, where* `while` *denotes the program fragment consisting of the* `while` *loop:*

$$\langle\langle \texttt{while} \rangle_{\mathsf{k}} \langle \texttt{a} \mapsto a\, \texttt{b} \mapsto b\, \texttt{x} \mapsto x\, \texttt{y} \mapsto y\, \texttt{r} \mapsto r \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge gcd(a,b) = gcd(x,y) \wedge x \geq 0 \wedge y \geq 0 \Rightarrow$$
$$\exists x', y', r'. \langle\langle \cdot \rangle_{\mathsf{k}} \langle \texttt{a} \mapsto a\, \texttt{b} \mapsto b\, \texttt{x} \mapsto x'\, \texttt{y} \mapsto y'\, \texttt{r} \mapsto r' \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge gcd(a,b) = gcd(x',y') \wedge x' \geq 0 \wedge y' \geq 0 \qquad (2)$$

procedure prove($\mathcal{S}, G$)
  if $G = \emptyset$ **then return** *true*
    else choose $\varphi \Rightarrow \varphi' \in G$
      if $\models \varphi \rightarrow \varphi'$ **then return** prove($\mathcal{S}, G \setminus \{\varphi \Rightarrow \varphi'\}$) // Implication Reduction
      else if there is $\alpha \in G_0$ that is applicable to $\varphi \Rightarrow \varphi'$ then
          return prove($\mathcal{S}, G \setminus \{\varphi \Rightarrow \varphi'\} \cup split(\Delta_\alpha(\varphi) \Rightarrow \varphi')$ // Circular Hypothesis
      else if φ is derivable for $\mathcal{S}$ **then**
          return prove($\mathcal{S}, G \setminus \{\varphi \Rightarrow \varphi'\} \cup split(\Delta_\mathcal{S}(\varphi) \Rightarrow \varphi')$ // Symbolic Step
      else return $false$.

Figure 5: Verification Procedure.

The rule says that the `while` *loop preserves an invariant: the gcd of the values of* `a`*,* `b` *equals the gcd of the values of* `x`*,* `y`*. We apply our procedure to the set of goals* G_0 *consisting of the formulas (1) and (2).*

- *the procedure first performs a number of recursive calls in the branch commented as Symbolic Step, until the lhs of the formula (1) in the current set of goals starts with a* `while` *instruction. This amounts to symbolically executing the code before the* `while` *instruction.*

- *then, a recursive call in the Circular Hypothesis branch is made, by which the goal (2) is used to derive a new goal, which at the next recursive call is discharged by the Implication Reduction branch.*

- *at this point, the current set of goals G consists of the singleton goal (2). The procedure makes a number of recursive calls in the Symbolic Step branch, until the current goal's lhs becomes a disjunction, due to the evaluation of the symbolic condition of an* `if` *statement (generated by the semantics of the* `while` *instruction in the goal (2)). By splitting the disjunction the current set of goals G now contains two new goals:*

  - *a first goal which contains an empty* `k` *cell, and which is discharged by a recursive call in the Implication Reduction branch;*
  - *a second goal, to which, after a few more calls in the Symbolic Step branch, a call in the Circular Hypothesis branch is made, reducing the currently singleton goal to one discharged by a subsequent call in the Implication Reduction branch.*

- *at this point, the current set of goals is empty. The deepest recursive call returns true, which is propagated upwards and eventually the initial call returns true as well.*

Remark 7 *For the soundness result it is not mandatory for the validity test* $\models \varphi \rightarrow \varphi'$ *to be exact; this is essential for implementation purposes, since exact validity checkers do not exist for first-order assertions due to their undecidability. The validity test can be implemented using SMT solvers, with the the only property that, whenever it answers "valid", the formula given to it as input is indeed valid. Specifically, the negation of the formula is given to the solver, and if the solver says the negation is unsatisfiable then the original formula is valid. The same holds for the "*α *covers* φ*" condition, which occurs in the procedure (in "*α *ia applicable to* $\varphi \Rightarrow \varphi'$*) and which is also a validity test.*

The weak completeness result, shown below, is concerned with situations when our procedure returns *false*. We need the following additional notions. A RL formula $\varphi \Rightarrow \varphi'$ is *terminating* if all configurations $\gamma \in [\![\varphi]\!] \cup [\![\varphi']\!]$ are terminating, and a set of formulas is terminating iff every formula in it is terminating. For example, the set consisting of formulas (1) and (2) is terminating. A semantics $\mathcal{S}$ is *confluent* if the rewrite rules in $\mathcal{S}$ are confluent in the usual sense. A semantics $\mathcal{S}$ is *live* if each right-hand side of a rule in $\mathcal{S}$ is derivable for $\mathcal{S}$ (that is, $\mathcal{S}$ does not artificially force programs to terminate).

Theorem 5 (Weak Completeness) *Consider a live and confluent set of RL formulas $\mathcal{S}$, a set of terminating formulas $G_0 = \{\pi_i \wedge \phi_i \Rightarrow \pi'_i \wedge \phi'_i | i \in I\}$, and a call $prove(S \cup G_0, split(\Delta_{\mathcal{S}}(G_0)))$ that returns false. Then, $\mathcal{S} \not\models G_0$.*

The set of formulas G in the above formulas must not have non-terminating instances and must be pairs of (unquantified) elementary patterns. In essence this says that for weak completeness to hold the program specification must be very precise: by disallowing quantifiers one needs to say exactly which values of variables are expected in the right-hand side, and by requiring terminating goals one needs to give strong enough side-conditions in patterns to ensure termination of all their instances. The liveness and confluence hypothesis on the semantics must also hold. Weak completeness moreover requires a sound *invalidity* test, which can also be implemented by SMT solvers. Together with the soundness result, weak completeness says that when our procedure terminates it correctly solves the program-verification problem given to it as input. Termination, of course, cannot be guaranteed, and it requires users to carefully choose the set of goals G_0 so that they can use each other as hypotheses in a circular manner during their respective proofs.

4 A Prototype Tool

In this section we describe our prototype `kcheck` that implements an iterative version of the (tail-recursive) procedure described in the previous section. We illustrate the tool on a parallel program and on a program implementing the Knuth-Morris-Pratt string-searching algorithm [21].

The `kcheck` tool is part of the $\mathbb{K}$ tool suite [2] and it has been developed on top of our language-independent symbolic execution tool [5]. $\mathbb{K}$ is a rewrite-based executable semantics framework in which programming languages, type systems, and formal analysis tools can be defined. Beside some toy languages used for teaching, there are a few real-word programming languages, supporting different paradigms, that have been successfully defined in $\mathbb{K}$, including Scheme [24], C [14], and Java [7]. An example of a $\mathbb{K}$ definition can be found in Section 2.2. In terms of implementation, our prototype reuses components of the $\mathbb{K}$ framework: parsing, compilation steps, support for symbolic execution, and connections to Maude's [23] state-space explorer and to the Z3 SMT solver [13]. We have used `kcheck` to prove `gcd.imp` (Figure 1) as sketched in Example 3. The tool has also been used to prove all the IMP programs from [4]. Since our approach is parametric in language definitions, it can be applied to programs from other $\mathbb{K}$ language definitions as well, as demonstrated in the following examples.

4.1 Verifying a parallel program: `FIND`

The example is inspired from [4]. Given an integer array a and a constant $N \geq 1$, the program in Figure 6 finds the smallest index $k \in \{1, \ldots, N\}$ such that $a[k] > 0$. If such an index k does not exists then $N + 1$ is returned. It is a *disjoint* parallel program, which means that its parallel components only have reading access to the variable a they share.

```
i = 1;                          S1 = while (i < oddtop) {
j = 2;                                 if (a[i] > 0) then { oddtop = i; }
oddtop = N + 1;                        else { i = i + 2; }
eventop = N + 1;                     }
S1 || S2;                       S2 = while (j < eventop) {
if (oddtop > eventop)                  if (a[j] > 0) then { eventop = j; }
  then { k = eventop; }                else { j = j + 2; }
  else { k = oddtop; }              }
```

Figure 6: `FIND` program.

In order to verify `FIND`, we have defined in $\mathbb{K}$ the semantics of a parallel language which provides assignments, if-statements, loops, arrays, dynamic threads, and the `||` operator, which executes in parallel two threads corresponding to `S1` and `S2`. In order to give to threads an access to their parent's variables we split the program state into an environment $\langle\rangle_{\mathsf{env}}$ and a store $\langle\rangle_{\mathsf{st}}$. An environment maps variable names into locations, while a store maps locations into values. Each thread $\langle\rangle_{\mathsf{th}}$ has its own computations $\langle\rangle_{\mathsf{k}}$ and environment $\langle\rangle_{\mathsf{env}}$ cells, while $\langle\rangle_{\mathsf{st}}$ is shared among the threads. Threads also have an $\langle\rangle_{\mathsf{id}}$ (identifier) cell. The configuration is shown below. The $^+$ on the $\langle\rangle_{\mathsf{th}}$ cell says that the cell contains at least one thread: $\langle\langle\langle \mathit{Code}\rangle_{\mathsf{k}}\langle \mathit{Map}_{\mathit{Id},\mathit{Int}}\rangle_{\mathsf{env}}\langle \mathit{Int}\rangle_{\mathsf{id}}\rangle_{\mathsf{th}^+}\langle \mathit{Map}_{\mathit{Int},\mathit{Int}}\rangle_{\mathsf{st}}\rangle_{\mathsf{cfg}}$. The `||` operator yields a non-deterministic behavior of `FIND`. However, in [4] the authors prove that the semantics of the language is confluent.For program verification this observation simplifies matters because it allows independent verification of the parallel code, without considering all the interleavings caused by parallelism.

The verification of `FIND` (see Appendix B) is performed by checking only three rules: one for each of the two loops and one for the main program. This is much simpler than the proof from [4], where more proof obligations must be generated and checked. Moreover, when using `kcheck` to verify `FIND`, we discovered that the precondition *pre* must be $N \geq 1$ rather than *true* as stated in the (non-mechanised) proof of [4], and in p_2 the value of j must be greater-or-equal to 2, a constraint that was also forgotten in [4].

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

The Knuth-Morris-Pratt algorithm [21] 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 can be found in [11]. Its code (in the CINK language [22], a fragment of C++ formally defined in $\mathbb{K}$) is shown in Figure 7.

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

The KMP algorithm uses a *failure function* π, which, for each position j in P, returns the length of the longest proper prefix of the pattern which is also a suffix of it. For our example, $\pi[3] = 1$ and $\pi[j] = 0$ for $j = 1, 2, 4$. In the case of a mismatch between the position i in T and the position j in P, the algorithm proceeds with the comparison of the positions i and $\pi[j]$. For the above mismatch, the next comparison is between the `B` in `ABAC` and the first instance of `D` in `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 7. The comments include the specifications

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

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

Figure 7: The KMP algorithm annotated with pre-/post-conditions and invariants: failure function (left) and the main function (right). Note that we used `Pi`, `Theta`, and `allOcc` to denote functions π and θ, and predicate *allOcc*, respectively.

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 [11]. Another formal proof of the algorithm is given in [15] by using Why3 [16]. 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 [11]. We also prove that KMP finds *all* the occurrences of the pattern in the string, not only the first one. We let $P[1..i]$ denote the prefix of P of size i, and $P[i]$ denote its i-th element.

Definition 8 *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:

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

$$\Rightarrow$$

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

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

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

For each annotated loop, the tool generates one RL formula, which states that by starting with a configuration where the entire loop remains to be executed and the loop invariant `INV` holds, one reaches a configuration where the loop was executed and `INV` $\wedge$ $\neg$`COND` holds.

From the annotations shown in Figure 7 the tool generates all the RL formulas that we need to prove KMP. Since KMP has four loops and two pairs of pre/post-conditions, the tool generates and proves a total number of six RL formulas. 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 enrich the symbolic definition of CINK with functional symbols for π, θ, and *allOcc*, we prove some of their properties independently in Coq [1], and use them in our verification with `kcheck`.

5 Conclusion, Related Work, and Future Work

We have presented a language-independent procedure and tool, based on symbolic execution, for automatically proving properties of programs expressed in Reachability Logic. With respect to the standard proof system of Reachability Logic our procedure can be seen as an automatic strategy for constructing proofs. The approach is partially correct, and the tool implementing it is illustrated on an parallel program example as well as on a complex string-matching program.

Related Work. There are several tools that perform program verification using symbolic execution. Some of them are more oriented towards finding bugs [8], while others are more oriented towards verification [10, 19, 25]. Several techniques are implemented to improve the

performance of these tools, such as *bounded verification* [9] and *pruning* the execution tree by eliminating redundant paths [12]. The major advantage of these tools is that they perform very well, being able to verify substantial pieces of C or assembly code, which are parts of actual safety-critical systems. On the other hand, these verifiers hardcode the logic they use for reasoning, and verify only specific programs (e.g. written using subsets of C) for specific properties such as, e.g., allocated memory is eventually freed.

Other approaches offer support for verification of code contracts over programs. Spec# [6] 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 call-backs 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 [17] and symbolic execution. The VeriFast tool [18] supports verification of single and multi-threaded C and Java programs annotated with preconditions and postconditions written in Separation Logic [26]. All these tools are designed to verify programs that belong to a specific programming language.

An approach closely related to ours is implemented in the MatchC tool [27], which has beed used for verifying several challenging C programs such as the Schorr-Waite garbage collector. MatchC also uses the RL formalism for program specifications; it is, however, dedicated to a specific programming language, and uses a particular implementation of the RL proof system whose correctness has not been formalized (to our best knowledge). By contrast, we focus on genericity, i.e., on *language-independence*: given a programming language defined in an algebraic/rewriting setting, we automatically generate the semantics for performing symbolic execution on that language, and build our proof system and its default program-verification strategy on the resulting symbolic execution engine. The correctness of our approach has also been proved.

Regarding performance, our generic tool is (understandably) not in the same league as tools targetting specific languages and/or specific program properties. We believe, however, that the building of fast language-specific verification tools can benefit from the general principles presented here, in particular, regarding the building of program-verification tools on top of symbolic execution engines.

Future Work. Reachability Logic, as a language-independent specification formalism, can be quite verbose and may not be easy to grasp by users who are more familiar to annotations *à la* Hoare logic (pre/post-conditions and invariants). Annotations are by definition language-specific since the statements that are annotated are specific to languages. However, common statements found in many languages (conditionals, loops, functions/procedures) can share the same annotations, from which RL formulas can be automatically generated. We are planning to explore this direction in order to improve the usability of our tool.

Another future research direction is making our verifier generate proof scripts for Coq [1], in order to obtain certificates that, despite any (inevitable) bugs in our tool, the proofs it generated are correct. This amounts to, firstly, encoding our procedure in Coq and proving its soundness. Secondly, `kcheck` must be instrumented to return an execution trace of our procedure. From this information a Coq script is built that, if sucessfully run by Coq, generates a proof term that constitutes a corectness certificate for the original `kcheck` execution.

References

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

[2] The $\mathbb{K}$ tool. `https://github.com/kframework/k`.

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

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

[5] A. Arusoaie, D. Lucanu, and V. Rusu. A generic framework for symbolic execution. In *6th International Conference on Software Language Engineering*, volume 8225 of *LNCS*, pages 281–301. Springer Verlag, 2013. Also available as a technical report `http://hal.inria.fr/hal-00853588`.

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

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

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

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

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

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

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

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

[14] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.

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

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

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

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

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

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

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

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

[23] J. Meseguer. Rewriting logic and Maude: Concepts and applications. In L. Bachmair, editor, *RTA*, volume 1833 of *LNCS*, pages 1–26. Springer, 2000.

[24] G. R. Patrick Meredith, Mark Hills. An Executable Rewriting Logic Semantics of K-Scheme. In D. Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701*, pages 91–103. Laval University, 2007.

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

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

[27] G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.

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

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

A Proofs

We say that two terms t_1, t_2 are concretely unifiable of there exists a valuation $\rho : \mathit{Var} \to \mathcal{T}$ such that $t_1\rho = t_2\rho$.

Lemma 1 *If t_1 and t_2 are terms such that t_1 is linear, has a non-data sort, and all its data subterms are variables; all the elements of $var(t_2)$ have data sorts; and t_1, t_2 are concretely unifiable, then there exists a substitution $\sigma_{t_2}^{t_1} : var(t_1) \mapsto T_\Sigma(var(t_2))$ such that $t_1\sigma_{t_2}^{t_1} = t_2$ and such that for all concrete unifiers ρ of t_1, t_2, there exists a valuation η such that $\rho = \sigma_{t_2}^{t_1}\eta$.*

Proof By induction on the structure of t_1. In the base case, $t_1 \in \mathit{Var}$ and $\sigma_{t_2}^{t_1} = (t_1 \mapsto t_2)$. Now, $\sigma_{t_2}^{t_1}$ is obviously such that $t_1\sigma_{t_2}^{t_1} = t_2$. To show the second conclusion of the lemma, consider any concrete unifier of t_1, t_2, say, ρ. Then, (a) $t_1\sigma_{t_2}^{t_1}\rho = t_2\rho$ because $\sigma_{t_2}^{t_1}$ maps t_1 to t_2, and (b) $t_2\rho = t_1\rho$ because ρ is a concrete unifier. Thus, $t_1\sigma_{t_2}^{t_1}\rho = t_1\rho$. Moreover, for all $x \in \mathit{Var} \setminus \{t_1\}$, $x\sigma_{t_2}^{t_1}\rho = x\rho$ since $\sigma_{t_2}^{t_1}$ only affects t_1. Thus, for all $y \in \mathit{Var}$, $y\sigma_{t_2}^{t_1}\rho = y\rho$, which proves the second conclusion of the lemma (by taking $\eta = \rho$).

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

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

  Let $\sigma_{t_2}^{t_1} \triangleq \biguplus_{i=1}^n \sigma_{t_2^i}^{t_1^i}$. We obtain $t_1\sigma_{t_2}^{t_1} = t_2$ from $t_1^i\sigma_{t_2^i}^{t_1^i} = t_2^i$ for all $i = 1, \ldots, n$. We only have to prove the second conclusion of the lemma. For this, we consider any concrete unifier ρ of t_1 and t_2, thus, $t_1^i\rho = t_2^i\rho$ for $i = 1, \ldots, n$. From the fact (obtained above) that all the $\sigma_{t_2^i}^{t_1^i}$ satisfy the second conclusion of the lemma, we obtain the existence of valuations η_i such that $\sigma_{t_2^i}^{t_1^i}\eta_i = \rho|_{var(t_1^i)}$, for $i = 1, \ldots, n$. Then, $\eta \triangleq \biguplus_{i=1}^n \eta_i$, has the property that $\sigma_{t_2}^{t_1}\eta = \rho$, which proves the second conclusion of the lemma and concludes the proof.

Remark 8 *The substitution $\sigma_{t_2}^{t_1}$ whose existence is stated by Lemma 1 is unique since it is a syntactical matcher of t_1 on t_2.*

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

Proof Let $\varphi \triangleq \pi \wedge \phi$. From $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ we obtain the rule $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ and the valuation $\rho : Var \to \mathcal{T}$ such that $\gamma = \pi_1\rho$, $\rho \models \phi_1$, $\rho \models \phi_2$, and $\gamma' = \pi_2\rho$. Without restriction of generality we can assume $var(\alpha) \cap var(\varphi) = \emptyset$, which can always be obtained by renaming variables. From $\gamma \in [\![\varphi]\!]$ we obtain the valuation $\rho' : Var \to \mathcal{T}$ such that $\gamma = \pi\rho'$ and $\rho' \models \phi$. Since $var(\alpha) \cap var(\varphi) = \emptyset$ we can take $\rho' = \rho$. Thus, π_1 and π are concretely unifiable (by their concrete unifier ρ). Using Lemma 1 we obtain the substitution $\sigma_\pi^{\pi_1} : var(\pi_1) \to T_\Sigma(var(\pi))$. Let then η be the valuation such that $\sigma_\pi^{\pi_1}\eta = \rho$. We extend $\sigma_\pi^{\pi_1}$ to Var by letting it be the identity on $Var \setminus var(\pi_1)$, and we can always choose η such that $\eta|_{Var \setminus var(\pi_1)} = \rho$. With these extensions we have $x(\sigma_\pi^{\pi_1}\eta) = x\rho$ for all $x \in Var$.

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

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

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

  Since the last relations hold by the hypotheses, it follows $\eta \models (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$. The following property was used above: if $\rho : Var \to \mathcal{T}$ is a valuation and $\sigma : Var \to T_\Sigma(Var)$ a substitution, then $\rho \models \varphi\sigma$ iff $\sigma\rho \models \varphi$.

The two above items imply $(\gamma', \eta) \models \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, i.e., $(\gamma', \eta) \models \varphi'$, which concludes the proof. The following theorem is a corollary to Lemma 2 and Assumption 1:

Theorem 1. *For every concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ and pattern φ_0 over variables of data sorts such that $\gamma_0 \in [\![\varphi_0]\!]$, there is a symbolic execution $\varphi_0 \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} \varphi_1 \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} \cdots \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} \varphi_n \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} \cdots$ such that φ_i only has variables of data sorts and $\gamma_i \in [\![\varphi_i]\!]$ for $i = 0, 1, \ldots$.*

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

Proof From $\varphi \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} \varphi'$ with $\varphi \triangleq \pi \wedge \phi$ and $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ we get $\varphi' = \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma$ for some matcher $\sigma : var(\pi_1) \to T_\Sigma(var(\pi))$ extended to the identify on $Var \setminus var(\pi_1)$. We assume without restriction of generality $var(\alpha) \cap var(\varphi) = \emptyset$.

From $\gamma' \in [\![\varphi']\!]$ we get $\eta : Var \to \mathcal{T}$ such that $\gamma' = (\pi_2\sigma)\eta$ and $\eta \models (\phi \wedge \phi_1 \wedge \phi_2)\sigma$. Let $\rho : Var \to \mathcal{T}$ be defined by $x\rho = x(\sigma\eta)$ for all $x \in var(\pi_1)$, and $x\rho = x\eta$ for all $x \in Var \setminus var(\pi_1)$, and let $\gamma \triangleq \pi_1\rho$. From $\gamma' = (\pi_2\sigma)\eta$ and the definition of ρ we obtain $\gamma' = \pi_2\rho$. From $\eta \models (\phi \wedge \phi_1 \wedge \phi_2)\sigma$ we get $\sigma\eta \models \phi_1$ and $\sigma\eta \models \phi_2$, i.e., $\rho \models \phi_1$ and $\rho \models \phi_2$, which together with $\gamma \triangleq \pi_1\rho$ and $\gamma' = \pi_2\rho$ gives $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. There remains to prove $\gamma \in [\![\varphi]\!]$.

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

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

We call a symbolic execution *feasible* if all its patterns are satisfiable (a pattern φ is satisfiable if there is a configuration γ such that $\gamma \in [\![\varphi]\!]$). The following theorem is a corollary to Lemma 3.

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

Before we prove the next lemma we need to recapp results about the original RL proof system, used in the sequel and proved in [27]:

- *Substitution*: $\mathcal{S} \vdash_G \varphi\theta \Rightarrow \varphi'\theta$, if $\theta : Var \to T_\Sigma(Var)$ and $\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'$;
- *Logical Framing*: $\mathcal{S} \vdash_G (\varphi \wedge \phi) \Rightarrow (\varphi' \wedge \phi)$, if ϕ is a patternless FOL formula and $\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'$;
- *Set Circularity*: if $\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'$ for each $\varphi \Rightarrow \varphi' \in G$ and G is finite then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ for each $\varphi \Rightarrow \varphi' \in G$;
- *Implication*: if $\models \varphi \to \varphi'$ then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$;
- *Monotony*: if $\mathcal{S} \subseteq \mathcal{S}'$ then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ implies $\mathcal{S}' \vdash \varphi \Rightarrow \varphi'$.

Lemma 4 *If $\mathcal{S}' \subseteq \mathcal{S}$ is a cover for φ, and G is a (possibly empty) set of RL formulas, then $\mathcal{S} \vdash_G \varphi \Rightarrow \Delta_{\mathcal{S}'}(\varphi)$.*

Proof Let $\varphi \triangleq \pi \wedge \phi$. By Definition 5, $\Delta_{\mathcal{S}'}(\varphi) \triangleq \bigvee_{\varphi \Rightarrow^{\mathsf{s}}_{\mathcal{S}'} \varphi'} \varphi'$. Since $\mathcal{S}'$ is a cover for φ, $\Delta_{\mathcal{S}'}(\varphi)$ is a nonempty disjunction. Using Definition 4 we obtain that each φ' is of the form $\varphi' = \pi_2\sigma_\pi^{\pi_1} \wedge \phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$ for some $\alpha \triangleq (\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in S$, where $\sigma_\pi^{\pi_1}$ is the substitution given by Lemma 1. By a variable renaming we can always assume that $var(\phi) \cap var(\pi_1) = \emptyset$, which means that the effect of $\sigma_\pi^{\pi_1}$ on ϕ is the identity, i.e., $\phi\sigma_\pi^{\pi_1} = \phi$.

Using the above characterisation for the patterns φ', we obtain

$$\Delta_{\mathcal{S}'}(\varphi) = \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1} \tag{3}$$

On the other hand, by using the derived rules of the RL proof system: *Substitution* with the rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ and substitution $\sigma_\pi^{\pi_1}$, and *LogicalFraming* with the patternless formula $\phi\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1}$, we get $\mathcal{S} \vdash_G (\pi_1 \wedge \phi_1)\sigma_\pi^{\pi_1} \wedge \phi\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1} \Rightarrow (\pi_2 \wedge \phi_2)\sigma_\pi^{\pi_1} \wedge \phi\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1}$. Using the Consequence rule of RL, and remembering that FOL patternless formulas distribute over patterns:

$$\mathcal{S} \vdash_G \pi\sigma_\pi^{\pi_1} \wedge (\phi\sigma_\pi^{\pi_1} \wedge \phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1}) \Rightarrow \pi_2\sigma_\pi^{\pi_1} \wedge (\phi\sigma_\pi^{\pi_1} \wedge \phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1})$$

Since the effect of $\sigma_\pi^{\pi_1}$ on both π and ϕ is the identity, we further obtain:

$$\mathcal{S} \vdash_G \pi \wedge (\phi \wedge \phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1}) \Rightarrow \pi_2\sigma_\pi^{\pi_1} \wedge (\phi\sigma_\pi^{\pi_1} \wedge \phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1})$$

Next, using CaseAnalysis and Consequence several times we obtain:

$$\mathcal{S} \vdash_G \pi \wedge \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} (\phi \wedge \phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1}) \Rightarrow \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1} \tag{4}$$

We know from (3) that the right-hand side of (4) is $\Delta_{\mathcal{S}'}(\varphi)$. To prove $\mathcal{S} \vdash_G \varphi \Rightarrow \Delta_{\mathcal{S}'}(\varphi)$ there only remains to prove ($\diamondsuit$): the condition in the left-hand side: $\bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} (\phi \wedge \phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1})$ is logically equivalent to ϕ in FOL. Since $\mathcal{S}'$ is a cover for φ, we obtain, using Definition 6, the validity of $\phi \to \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} (\phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1})$, which proves ($\diamondsuit$) and the lemma.

The next lemma is "almost" the Circularity Principle for RL (Theorem 3) except that it contains an additional hypothesis (weak well-definedness of the semantics). We will later see that this hypothesis is redundant.

Lemma 5 *If $\mathcal{S}$ is total and weakly well-defined, and G is derivable for $\mathcal{S}$, then $\mathcal{S} \cup G \vdash \Delta_{\mathcal{S}}(G)$ implies $\mathcal{S} \models G$.*

Proof For all $i = 1, \ldots, n$ we apply the Transitivity rule of the original RL proof system, with $\varphi_i'' \triangleq \Delta_{\mathcal{S}}(\varphi_i)$, and obtain:

$$\frac{\mathcal{S} \vdash_G \varphi_i \Rightarrow \Delta_{\mathcal{S}}(\varphi_i) \quad (\mathcal{S} \cup G) \vdash \Delta_{\mathcal{S}}(\varphi_i) \Rightarrow \varphi_i'}{\mathcal{S} \vdash_G \varphi_i \Rightarrow \varphi_i'}$$

The first hypothesis: $S \vdash_G \varphi_i \Rightarrow \Delta_{\mathcal{S}}(\varphi_i)$ holds thanks to Lemma 4 and the totality of $\mathcal{S}$. The second one, $(\mathcal{S} \cup G) \vdash \Delta_{\mathcal{S}}(\varphi_i) \Rightarrow \varphi_i'$ holds by hypothesis. Hence, we obtain $S \vdash_G \varphi_i \Rightarrow \varphi_i'$ for all $i = 1, \ldots, n$, i.e., $S \vdash_G G$. Then we obtain $S \vdash G$ by applying the derived rule *Set Circularity* of RL. Finally, the soundness of $\vdash$ (with the hypothesis that $\mathcal{S}$ is weakly well defined) implies $\mathcal{S} \models G$, which concludes the proof.

We now show how the weak well-definedness hypothesis can be eliminated. We first note as an example that the semantics of IMP is not weakly well-defined, because of the rules for division and modulo that do not have instances for valuations mapping I_2 to 0.

However, due to the introduction of the rule $\langle\langle I_1 \,\%\, I_2 \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}} \wedge I_2 =_{Int} 0 \Rightarrow \langle error\rangle_{\mathsf{cfg}}$ in order to make the semantics total (cf. Remark 6), the semantics of division can now equivalently rewritten using just one (reachability-logic) disjunctive rule:

$$\langle\langle I_1 \,\%\, I_2 \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0 \Rightarrow (\langle\langle I_1 \%_{Int} I_2 \,\cdots\rangle_{\mathsf{k}} \,\cdots\rangle_{\mathsf{cfg}} \wedge I_2 =_{Int} 0) \vee (\langle error\rangle_{\mathsf{cfg}})$$

By using this rule instead of the two original ones, and by applying the same transformation for the rules defining division, the semantics becomes both total and weakly well-defined. This transformation is formalised as follows.

Definition 9 ($\mathcal{S}^\Delta$) *Given a set of semantical rules $\mathcal{S}$, the set of semantical rules $\mathcal{S}^\Delta$ is defined by $\mathcal{S}^\Delta \triangleq \{\pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \mid (\pi \wedge \phi \Rightarrow \varphi') \in \mathcal{S}\}$.*

The following lemma establishes that $\mathcal{S}^\Delta$ has both properties (totality and weak well definedness) required for the soundness of our approach. For this, we need extend the notion of derivative for rules of the form $\mathcal{S}^\Delta$ (containing disjunctions in their right-hand side) by letting $\Delta(\mathcal{S}^\Delta)(\varphi) \triangleq \Delta(\varphi)$. The derivability of a pattern for $\mathcal{S}^\Delta$ is also, by definition, the derivability for $\mathcal{S}$. The notion of cover is extended for such rules, of the form $\pi_1 \wedge \phi_1 \Rightarrow \bigvee_{j \in J} \pi_2^j \wedge \phi_2^j$ by letting $\varphi \triangleq \pi \wedge \phi$ be covered by a set $\mathcal{S}'$ of such rules if $\models \varphi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \bigvee_{j \in J} \pi_2^j \wedge \phi_2^j \in \mathcal{S}'} \bigvee_{j \in J} (\phi_1 \wedge \phi_j^2) \sigma_\pi^{\pi^1}$.

Lemma 6 *If $\mathcal{S}$ is total then $\mathcal{S}^\Delta$ is total and weakly well-defined.*

Proof

- *Totality of $\mathcal{S}^\Delta$:* If $\mathcal{S}$ is total then, by definition, $\mathcal{S}$ covers all patterns φ derivable for $\mathcal{S}$.

  We need to prove that for all φ that is derivable for $\mathcal{S}^\Delta$, $\mathcal{S}^\Delta$ is a cover for φ. Assume such a $\varphi \triangleq \pi \wedge \phi$ derivable for $\mathcal{S}^\Delta$. We have defined derivability for $\mathcal{S}^\Delta$ as derivability for $\mathcal{S}$, hence φ is derivable for $\mathcal{S}$. The totality of $\mathcal{S}$ and the definition of cover then ensures $\models \phi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} (\phi_1 \wedge \phi_2) \sigma_\pi^{\pi_1}$.

  Now, the rules in $\mathcal{S}^\Delta$ are of the form $\pi \Rightarrow \Delta_{\mathcal{S}}(\pi)$, for some π occurring in the pattern LHS of a rule in $\mathcal{S}$. We have $\Delta_{\mathcal{S}}(\pi) = \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \pi_2 \sigma_\pi^{\pi_1} \wedge (\phi_1 \wedge \phi_2) \sigma_\pi^{\pi_1}$ (implicitly, the disjunction performed over those rule in $\mathcal{S}$ for which $\sigma_\pi^{\pi^1}$ exists). Hence, by the notion of cover applied to $\mathcal{S}^\Delta$, in order to show that $\mathcal{S}^\Delta$ is total, we need to show

$$\models \phi \rightarrow \bigvee_{(\pi \Rightarrow \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}} \pi_2 \sigma_\pi^{\pi_1} \wedge (\phi_1 \wedge \phi_2) \sigma_\pi^{\pi_1}) \in \mathcal{S}^\Delta} \quad \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} (\phi_1 \wedge \phi_2) \sigma_\pi^{\pi_1} \qquad (5)$$

in which the second disjunction $\bigvee_{\pi_1\wedge\phi_1\Rightarrow\pi_2\wedge\phi_2\in\mathcal{S}}(\phi_1\wedge\phi_2)\sigma_\pi^{\pi_1}$ is merely repeated by the first disjunction $\bigvee_{(\ldots)\in\mathcal{S}^\Delta}$ as many times as rules in $\mathcal{S}^\Delta$. Hence, (5) simplifies to $\models \phi \rightarrow \bigvee_{\pi_1\wedge\phi_1\Rightarrow\pi_2\wedge\phi_2\in\mathcal{S}}(\phi_1\wedge\phi_2)\sigma_\pi^{\pi_1}$, which he have obtained above.

- *Weak well-definedness of $\mathcal{S}^\Delta$:* we need to show that for $\pi \Rightarrow \Delta_\mathcal{S}(\pi) \in \mathcal{S}^\Delta$ and each valuation ρ, there exists a configuration γ such that $(\gamma,\rho) \models \Delta_S(\pi)$. We have obtained above that $\Delta_\mathcal{S}(\pi) = \bigvee_{\pi_1\wedge\phi_1\Rightarrow\pi_2\wedge\phi_2\in\mathcal{S}} \pi_2\sigma_\pi^{\pi_1} \wedge (\phi_1\wedge\phi_2)\sigma_\pi^{\pi_1}$ (where implicitly the disjunction is performed over those rule in $\mathcal{S}$ for which $\sigma_\pi^{\pi_1}$ exists). Now, $\mathcal{S}$ is total, and $\pi(=\pi\wedge true)$ is derivable for $\mathcal{S}$. This implies $\models true \rightarrow \bigvee_{\pi_1\wedge\phi_1\Rightarrow\pi_2\wedge\phi_2\in\mathcal{S}}(\phi_1\wedge\phi_2)\sigma_\pi^{\pi_1}$, i.e., $\models \bigvee_{\pi_1\wedge\phi_1\Rightarrow\pi_2\wedge\phi_2\in\mathcal{S}}(\phi_1\wedge\phi_2)\sigma_\pi^{\pi_1}$. This means that for any ρ there exists in the above disjunction at least one disjunct, say, $(\phi_1^i\wedge\phi_2^i)\sigma_\pi^{\pi_1}$, such that $\rho \models (\phi_1^i\wedge\phi_2^i)\sigma_\pi^{\pi_1}$. By taking the corresponding $\gamma \triangleq \pi_2\sigma_\pi^{\pi_1}\rho$ we obtain $(\gamma,\rho) \models \pi_2\sigma_\pi^{\pi_1} \wedge (\phi_1^i\wedge\phi_2^i)\sigma_\pi^{\pi_1}$, i.e., (γ,ρ) models one of the disjuncts of $\Delta_\mathcal{S}(\pi)$. This implies $(\gamma,\rho) \models \Delta_S(\pi)$, which completes the proof.

The next lemma says that the transition systems $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}^\Delta})$ and $(\mathcal{T}_{Cfg}, \Rightarrow_\mathcal{S})$ are the same:

Lemma 7 *$\gamma \Rightarrow_{\mathcal{S}^\Delta} \gamma'$ if and only $\gamma \Rightarrow_\mathcal{S} \gamma'$.*

Proof ($\Rightarrow$) $\gamma \Rightarrow_{\mathcal{S}^\Delta} \gamma'$ means that there exists a rule in $\mathcal{S}^\Delta$, say, $\pi \Rightarrow \Delta_\mathcal{S}(\pi)$ obtained from some rule $\pi\wedge\phi \Rightarrow \varphi' \in \mathcal{S}$ such that $(\gamma,\rho) \models \pi$ and $(\gamma',\rho) \models \Delta_\mathcal{S}(\pi)$. We have $\Delta_\mathcal{S}(\pi) = \bigvee_{\pi_1\wedge\phi_1\Rightarrow\pi_2\wedge\phi_2\in\mathcal{S}} \pi_2\sigma_\pi^{\pi_1} \wedge (\phi_1\wedge\phi_2)\sigma_\pi^{\pi_1}$ (implicitly, the disjunction is performed over those rule in $\mathcal{S}$ for which $\sigma_\pi^{\pi_1}$ exists). Thus, $(\gamma',\rho) \models \Delta_\mathcal{S}(\pi)$ implies that there exists $\pi_1\wedge\phi_1 \Rightarrow \pi_2\wedge\phi_2 \in \mathcal{S}$ such that $\sigma_\pi^{\pi_1}$ exists and $(\gamma',\rho) \models \pi_2\sigma_\pi^{\pi_1}\wedge(\phi_1\wedge\phi_2)\sigma_\pi^{\pi_1}$ holds. Let $\eta \triangleq \sigma_\pi^{\pi_1}\rho$, then we obtain $(\gamma',\eta) \models \pi_2\wedge\phi_2$ and $\eta \models \varphi_1$. On the other hand, $(\gamma,\rho) \models \pi = \pi_1\sigma_\pi^{\pi_1}$ because $\sigma_\pi^{\pi_1}$ a matcher, and we obtain $(\gamma,\eta) \models \pi_1$ as well. Hence, we have the transition $\gamma \Rightarrow_\mathcal{S} \gamma'$ generated by the rule $\pi_1\wedge\phi_1 \Rightarrow \pi_2\wedge\phi_2 \in \mathcal{S}$ and the valuation η, which proves the ($\Rightarrow$) implication.

($\Leftarrow$) $\gamma \Rightarrow_\mathcal{S} \gamma'$ means there exists $\pi_1\wedge\phi_1 \Rightarrow \pi_2\wedge\phi_2 \in \mathcal{S}$ and a valuation ρ such that $(\gamma,\rho) \models \pi_1\wedge\phi_1$ and $(\gamma',\rho) \models \pi_2\wedge\phi_2$. Consider the rule $\pi_1 \Rightarrow \Delta_\mathcal{S}(\pi_1) \in \mathcal{S}^\Delta$ (obtained from $\pi_1\wedge\phi_1 \Rightarrow \pi_2\wedge\phi_2 \in \mathcal{S}$). Obviously, $\sigma_{\pi_1}^{\pi_1}$ exists (it is the identity) and we have $(\gamma',\rho) \models \pi_2\sigma_{\pi_1}^{\pi_1}\wedge(\phi_1\wedge\phi_2)\sigma_{\pi_1}^{\pi_1}$, which is a disjunct of $\Delta_\mathcal{S}(\pi_1)$, implying that $(\gamma',\rho) \models \Delta_\mathcal{S}(\pi_1)$. From the latter and $(\gamma,\rho) \models \pi_1\wedge\phi_1$ we obtain the transition $\gamma \Rightarrow_{\mathcal{S}^\Delta} \gamma'$ generated by $\pi_1 \Rightarrow \Delta_\mathcal{S}(\pi_1) \in S^\Delta$ and the valuation ρ, which proves the ($\Leftarrow$) implication and the lemma.

Theorem 3 (Circularity Principle for RL). *If $\mathcal{S}$ is total and G is derivable for $\mathcal{S}$, then $\mathcal{S}\cup G \vdash \Delta_\mathcal{S}(G)$ implies $\mathcal{S} \models G$.*

Proof By Lemma 6 and the totality of $\mathcal{S}$ is both total and weakly well-defined. By definition derivability w.r.t. S^Δ and $\mathcal{S}$ are the same, hence, we can apply Lemma 5 with $\mathcal{S} \triangleq \mathcal{S}^\Delta$ and obtain that $\mathcal{S}\cup G \vdash \Delta_{\mathcal{S}^\Delta}(G)$ implies $\mathcal{S}^\Delta \models G$. However, we have defined $\Delta_{\mathcal{S}^\Delta}(G)$ to be $\Delta_\mathcal{S}(G)$, hence, if we prove that $\mathcal{S}^\Delta \models G$ iff $\mathcal{S} \models G$ then the theorem is proved. For this, use use Lemma 7 to show that (i) a configuration γ is terminating in the transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}^\Delta})$ iff γ is terminating in $(\mathcal{T}_{Cfg}, \Rightarrow_\mathcal{S})$, and (ii) ($(\gamma,\rho) \models \varphi$ *implies that there exists* $\gamma \Rightarrow_\mathcal{S}^* \gamma'$ *such that* $(\gamma',\rho) \models \varphi'$ if and only if the same implication holds in $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}^\Delta})$, i.e., $(\gamma,\rho) \models \varphi$ *implies that there exists* $\gamma \Rightarrow_{\mathcal{S}^\Delta}^* \gamma'$ *such that* $(\gamma',\rho) \models \varphi'$. (i) and (ii) mean, by definition, that $\mathcal{S}^\Delta \models G$ iff $\mathcal{S} \models G$, which concludes the proof.

We now proceed towards proving the soundness result. As an intermediary step, we introduce a proof system and prove its soundness. Then we show that our procedure is an implementation of the proof system.

Lemma 8 *Consider the proof system $\vdash$ in Figure 8. If $\mathcal{S}$ is total, then $\mathcal{S}\cup G \Vdash \varphi \Rightarrow \varphi'$ implies $\mathcal{S}\cup G \vdash \varphi \Rightarrow \varphi'$.*

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

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

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

$$\text{[Split]}\ \frac{\mathcal{S} \cup G \Vdash \mathit{split}(\varphi \Rightarrow \varphi')}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \varphi'}$$

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

Figure 8: Proof System $\Vdash$.

Proof We show that every rule in the proof system in Figure 8 is a derived rule w.r.t. the original proof system of RL. For the SymbolicStep and CircularHypothesis rules this is a direct consequence of Lemma 4. The ImplicationReduction rule is obtained by combining the Consequence and Reflexivity rules of the original proof system. Split consists in (possibly, several) applications of theCaseAnalysis of the original proof system and Transitivity is a particular case of the homonymous rule in the original proof system (without circular hypotheses - our verification method uses the Circular Principle instead).

We are now ready to prove the soundness and weak completeness results. They concern our verification procedure, which we reproduce below.

procedure prove($\mathcal{S}, G$)
  if $G = \emptyset$ **then return** *true*
    else choose $\varphi \Rightarrow \varphi' \in G$
      if $\models \varphi \rightarrow \varphi'$ **then return** prove($\mathcal{S}, G \setminus \{\varphi \Rightarrow \varphi'\}$) // Implication Reduction
      else if there is $\alpha \in G_0$ such that α is applicable to $\varphi \Rightarrow \varphi'$ **then**
          return prove($\mathcal{S}, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \mathit{split}(\Delta_\alpha(\varphi) \Rightarrow \varphi')$ // Circular Hypothesis
      else if φ is derivable for $\mathcal{S}$ **then**
          return prove($\mathcal{S}, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \mathit{split}(\Delta_{\mathcal{S}}(\varphi) \Rightarrow \varphi')$ // Symbolic Step
      else return $false$.

Theorem 4 (soundness). *Assume that for a total semantics $\mathcal{S}$ and a set of goals G_0 derivable for $\mathcal{S}$, the call prove($S \cup G_0, \mathit{split}(\Delta_{\mathcal{S}}(G_0))$) returns true. Then $\mathcal{S} \models G_0$.*

Proof Let $\mathcal{S}' \triangleq \mathcal{S} \cup G_0$. We consider more generally any call $\mathit{prove}(\mathcal{S}', G')$ that returns *true*, and define a relation *less than* on recursive calls to *prove*() generated by the call $\mathit{prove}(\mathcal{S}', G')$, by the fact that any call is *less than* the call that directly generated it. Since the call $\mathit{prove}(\mathcal{S}', G')$ terminates the relation *less than* is well founded (i.e., there is no infinite chain of recursive calls in which each call is *less than* its predecessor). We prove by well-founded induction on this relation that ($\diamondsuit$) if $\mathit{prove}(\mathcal{S}', G') = \mathit{true}$ then $\mathcal{S}' \Vdash G'$. Now, $\mathit{prove}(\mathcal{S}', G') = \mathit{true}$ may only result from the following situations:

- either $G' = \emptyset$, in which case $\mathcal{S}' \Vdash G'$ holds trivially;

- or the recursive call $prove(\mathcal{S}', G' \setminus \{\varphi \Rightarrow \varphi'\})$ returns *true*: then, we apply the induction hypothesis and obtain $\mathcal{S}' \Vdash G' \setminus \{\varphi \Rightarrow \varphi'\}$, and using the Implication rule we obtain from $\models \varphi \rightarrow \varphi'$ that $\mathcal{S}' \Vdash \varphi \Rightarrow \varphi'$, which gives us $\mathcal{S}' \Vdash G'$;

- or the recursive call $prove(\mathcal{S}', G' \setminus \{\varphi \Rightarrow \varphi'\} \cup split(\Delta_\alpha(\varphi) \Rightarrow \varphi')$ returns *true*. Using the induction hypothesis we obtain $\mathcal{S}' \Vdash G' \setminus \{\varphi \Rightarrow \varphi'\} \cup split(\Delta_\alpha(\varphi) \Rightarrow \varphi')$. In particular, $\mathcal{S}' \Vdash split(\Delta_\alpha(\varphi) \Rightarrow \varphi')$, which thanks to the Split rule of the $\Vdash$ system implies $\mathcal{S}' \Vdash \Delta_\alpha(\varphi) \Rightarrow \varphi'$. Using the CircularHypothesis rule (which can be used, since in this case α is applicable to $\varphi \Rightarrow \varphi'$ which implies by definition that α covers φ) we obtain $\mathcal{S}' \Vdash \varphi \Rightarrow \Delta_\alpha(\varphi)$ and using the Transitivity rule we obtain $\mathcal{S}' \Vdash \varphi \Rightarrow \varphi'$, which together with $\mathcal{S}' \Vdash G' \setminus \{\varphi \Rightarrow \varphi'\}$ gives us $\mathcal{S}' \Vdash G'$;

- or the recursve call $prove(\mathcal{S}', G' \setminus \{\varphi \Rightarrow \varphi'\} \cup split(\Delta_\mathcal{S}(\varphi) \Rightarrow \varphi')$ returns *true*. We obtain by induction hypothesis that $\mathcal{S}' \Vdash G' \setminus \{\varphi \Rightarrow \varphi'\} \cup split(\Delta_\mathcal{S}(\varphi) \Rightarrow \varphi')$. In particular, $\mathcal{S}' \Vdash split(\Delta_S(\varphi) \Rightarrow \varphi')$, which thanks to the Split rule of the $\Vdash$ system gives us $\mathcal{S}' \Vdash \Delta_\mathcal{S}(\varphi) \Rightarrow \varphi'$. Using the SymbolicStep rule we obtain $\mathcal{S}' \Vdash \varphi \Rightarrow \Delta_\mathcal{S}(\varphi)$ and using the Transitivity rule we obtain $\mathcal{S}' \Vdash \varphi \Rightarrow \varphi'$, which together with $\mathcal{S}' \Vdash G' \setminus \{\varphi \Rightarrow \varphi'\}$ gives us $\mathcal{S}' \Vdash G'$.

$(\diamondsuit)$ is now proved. By letting $G' \triangleq \Delta_\mathcal{S}(G_0)$ in $(\diamondsuit)$, and remembering that we denoted $\mathcal{S}' = \mathcal{S} \cup G_0$, we obtain $\mathcal{S} \cup G_0 \Vdash \Delta_\mathcal{S}(G_0)$. By using Lemma 8 we obtain $\mathcal{S} \cup G_0 \vdash \Delta_\mathcal{S}(G_0)$, which, by Theorem 3 implies $S \models G_0$, which proves our theorem.

Theorem 5 (weak completeness). *Consider a live and confluent set of RL formulas $\mathcal{S}$, a set of terminating formulas $G_0 = \{\pi_i \wedge \phi_i \Rightarrow \pi_i' \wedge \phi_i' | i \in I\}$, and a call $prove(S \cup G_0, split(\Delta_\mathcal{S}(G_0)))$ that returns false. Then, $\mathcal{S} \not\models G_0$.*

Proof We first prove by induction on the number of calls that in any recursive call generated by $prove(\mathcal{S} \cup G_0, \Delta_\mathcal{S}(G_0))$, and any formula $\varphi \Rightarrow \varphi' \in G$, the elementary pattern φ is a disjunct of $\Delta_S(\Delta^*_{\mathcal{S} \cup G_0}(\varphi_0))$, for φ_0 being the lhs of a goal in G_0. This holds because at each recursive call, some goals are unmodified, while the others are derived with respect to $\mathcal{S}$ or to G_0.

From this we obtain that there exists a symbolic execution $\varphi_0 \Rightarrow^{\mathfrak{s}}_\mathcal{S} \varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S} \cup G_0} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S} \cup G_0} \varphi$.

Next, we prove that the set of goals G is terminating at each recursive call, also by induction on the number of calls. This holds initially, and then, for each formula $\varphi \Rightarrow \varphi'$ in the current set of goals, the rhs φ' is the rhs of a goal in the original set of goals G_0 (since our procedure does not modify right-hand sides of formulas). Concerning the lhs φ, it is:

- either a disjunct of $\Delta_\mathcal{S}(\varphi'')$ where φ'' is terminating (by the induction hypothesis). Assuming there is a nonterminating $\gamma \in [\![\varphi]\!]$, using Lemma 3 we would obtain a concrete transition $\gamma'' \Rightarrow_\mathcal{S} \gamma$ with $\gamma'' \in [\![\varphi'']\!]$, hence φ'' would also be nonterminating, a contradiction;

- or φ is a disjunct of $\Delta_{\{\alpha\}}(\varphi'')$ for some $\alpha \in G_0$. But then all $\gamma \in [\![\varphi]\!]$ are also instances of the rhs of α, which is terminating by hypothesis, hence, φ is terminating as well.

Hence, the current set of goals G is terminating. We now choose a formula $\varphi \Rightarrow \varphi' \in G$ such that the procedure directly returns *false* while processing $\varphi \Rightarrow \varphi'$ in the procedure's loop (i.e., there are no further recursive calls). We thus have $\not\models \varphi \rightarrow \varphi'$, hence, we can choose $\gamma \in [\![\varphi]\!] \setminus [\![\varphi']\!]$. The above-obtained (feasible) symbolic execution $\varphi_0 \Rightarrow^{\mathfrak{s}}_\mathcal{S} \varphi_1 \Rightarrow^{\mathfrak{s}}_{\mathcal{S} \cup G_0} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S} \cup G_0} \varphi$ is simulated by a concrete execution $\gamma_0 \Rightarrow_\mathcal{S} \gamma_1 \Rightarrow^*_{\mathcal{S} \cup G_0} \gamma$ with $\gamma_0 \in [\![\varphi_0]\!]$, $\gamma_1 \in [\![\varphi_1]\!]$ thanks to Lemma 3.

Assume now (by contradiction) $S \models G_0$.

First, we show $(\diamondsuit)$: $\gamma_0 \Rightarrow^*_\mathcal{S} \gamma$. For this, we show that for every step, say, $\gamma_i \Rightarrow_{\{\alpha\}} \gamma_{i+1}$ with $\alpha \in G_0$, there is an execution with rules in $\mathcal{S}$, i.e., $\gamma_i \Rightarrow^*_\mathcal{S} \gamma_{i+1}$. Then we replace every such step

$\gamma_i \Rightarrow_{\{\alpha\}} \gamma_{i+1}$ in $\gamma_0 \Rightarrow^*_{\mathcal{S}\cup G_0} \gamma$ by the corresponding execution $\gamma_i \Rightarrow^*_{\mathcal{S}} \gamma_{i+1}$ and obtain the desired execution $\gamma_0 \Rightarrow^*_{\mathcal{S}} \gamma$. We now prove (♠) $\gamma_i \Rightarrow^*_{\mathcal{S}} \gamma_{i+1}$ from $\gamma_i \Rightarrow_{\{\alpha\}} \gamma_{i+1}$ and $\alpha \in G_0$.

From the assumption $\mathcal{S} \models G_0$ we obtain $\mathcal{S} \models \alpha (= \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in G_0)$.

From $\gamma_i \Rightarrow_{\{\alpha\}} \gamma_{i+1}$ we get $(\gamma_i, \rho) \models \pi_1 \wedge \phi_1$ for some valuation ρ and $(\gamma_{i+1}, \rho) \models \pi_2 \wedge \phi_2$. Thus, $\gamma_{i+1} = \pi_2\rho$. From $\mathcal{S} \models \alpha$ and $(\gamma_i, \rho) \models \pi_1 \wedge \phi_1$ (note that γ_i is terminating, since G_0 is terminating) we get that there exists γ'_{i+1} such that $\gamma_i \Rightarrow^*_{\mathcal{S}} \gamma'_{i+1}$ and $(\gamma'_{i+1}, \rho) \models \pi_2 \wedge \phi_2$. In particular, $\gamma'_{i+1} = \pi_2\rho$. Thus, $\gamma'_{i+1} = \gamma_{i+1}$, so $\gamma_i \Rightarrow^*_S \gamma_{i+1}$. (♠) is now proved, and so is (♢).

We thus have $\gamma_0 \Rightarrow^*_{\mathcal{S}} \gamma$, where $\gamma \in [\![\varphi]\!] \setminus [\![\varphi']\!]$, and γ is terminating (since $\gamma \in [\![\varphi]\!]$ and G is terminating as established earlier in the proof). On the other hand, we know from the beginning of the proof that φ is a disjunct of $\Delta_S(\Delta^*_{\mathcal{S}\cup G_0}(\varphi_0))$, for φ_0 the lhs of a goal in G_0. Since our procedure does not modify right-hand sides of goals, we obtain that $\varphi_0 \Rightarrow \varphi'$ is a goal of G_0.

We have assumed $\mathcal{S} \models G_0$, in particular, $\mathcal{S} \models \varphi_0 \Rightarrow \varphi'$. This means that from the (terminating) $\gamma_0 \in [\![\varphi_0]\!]$, there is $\gamma' \in [\![\varphi']\!]$ and $\gamma_0 \Rightarrow^*_{\mathcal{S}} \gamma'$. Since $\gamma \in [\![\varphi]\!] \setminus [\![\varphi']\!]$ and $\gamma' \in [\![\varphi']\!]$ we have $\gamma \neq \gamma'$.

Note that tφ is not derivable for $\mathcal{S}$, since otherwise the procedure would not be returning *false* on $\varphi \Rightarrow \varphi'$ but would go on a recursive call There are now two cases:

- if φ' is not derivable for $\mathcal{S}$: then, there is no way that γ and γ', which are distinct and are both successors of γ_0, may have a common successor, since both φ and φ' are non-derivable for $\mathcal{S}$. This contradicts the confluence hypothesis;
- if φ' is derivable for $\mathcal{S}$: Now, the non-derivable φ cannot be a disjoint of the goals G_0 since the goals are derivable for $\mathcal{S}$. φ cannot have been generated by earlier recursive call which applied a rule $\alpha \in G_0$ because such rules cannot be applied to generate a non-derivable pattern φ when φ' is derivable (the "applicability" condition in the algorithm prevents this). And finally, φ cannot have been generated by a rule in $\mathcal{S}$ since by hypothesis, $\mathcal{S}$ is live, i.e., all the right-hand sides of rules in $\mathcal{S}$ are derivable for $\mathcal{S}$. Hence, this case is impossible.

All cases lead to contradictions, generated by our assumption $\mathcal{S} \models G_0$: we conclude $\mathcal{S} \not\models G_0$.

B Verification of the program FIND

Figure 7 shows all the ingredients that we used to prove the correctness of the program `FIND` (Figure 6) using our tool. At the figure's top we show the code macros that we use in our RL formulas. Below the code macros we include the formulas corresponding to the pre/post conditions and invariants used by the authors of [4] in their proof. The program is checked by applying the implementation `kcheck` of our proof system on the consisting of the three RL formula-set $G = \{(\clubsuit), (\diamondsuit), (\spadesuit)\}$. On the bottom lines we show the proofs automatically constructed by `kcheck`.

We believe that the number of three proof obligations, given by G, is minimal for verifying `FIND`. Initially we started we eight rules describing the proof obligations used in [4]. Then, based on the `gcd` examples and others inspired from the same source, we realised that all sequential program fragment specifications can be removed since they can be automatically proved using the SymbolicStep rule, which amounts to symbolic execution. Since the configuration for the new language is more complex, the syntax for these rules is a bit cumbersome, but it can be generated from the annotations of the program by using symbolic execution to determine the exact structure of the configuration at the point where such a rule should be applied. We are developing a tool intended to help the user in writing these rules.

The proof trees for the RL formulas ($\clubsuit$) and ($\diamondsuit$) are similar to that of (2) for the `gcd` program. However, here the second branch is splitted by a new use of the CaseAnalysis rule, due to the `if` statement from the loop's body. The proof tree for the RL formula ($\spadesuit$), corresponding to the specification of `FIND`, has a single branch because it uses circularities ($\clubsuit$) and ($\diamondsuit$) that do not split the proof tree.

The formulas are nontrivial, and it took us several iterations to come up with the exact ones, during which we used the tool in a trial-and-error process. The automatic nature of the tool, as well as the feedback it returned when it failed, were particularly helpful during this process. In particular symbolic execution was fruitfully used for the initial testing of programs before they were verified.

```
i = 1;
j = 2;
oddtop = N + 1;
eventop = N + 1;
S1 || S2;
if (oddtop > eventop)
  then { k = eventop; }
  else { k = oddtop; }
```

```
S1 = while (i < oddtop) {
       if (a[i] > 0) then { oddtop = i; }
       else { i = i + 2; }
     }
S2 = while (j < eventop) {
       if (a[j] > 0) then { eventop = j; }
       else { j = j + 2; }
     }
```

Figure 6: `FIND` program.

CODE MACROS		
`INIT`	$\triangleq$	`i = 1; j = 2; oddtop = N + 1; eventop = N + 1;`
`BODY1`	$\triangleq$	`{if (a[i] > 0) then { oddtop = i; } else { i = i + 2; }}`
`BODY2`	$\triangleq$	`{if (a[j] > 0) then { eventop = j; } else { j = j + 2; }}`
`S1`	$\triangleq$	`while (i < oddtop) BODY1`
`S2`	$\triangleq$	`while (j < eventop) BODY2`
`MIN`	$\triangleq$	`if (oddtop > eventop) then { k = eventop; } else { k = oddtop; }`
`FIND`	$\triangleq$	`INIT S1||S2; MIN`

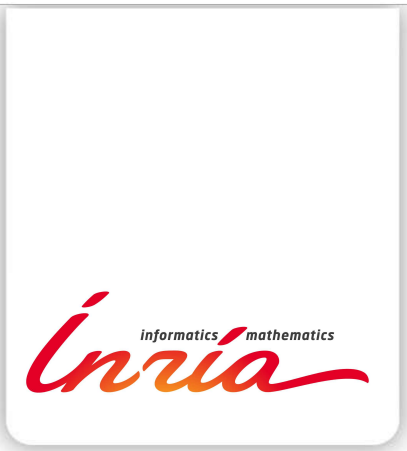
Formula macros		
pre	$\triangleq$	$N \geq 1$
p_1	$\triangleq$	$1 \leq o \leq N+1 \wedge i\%2 = 1 \wedge 1 \leq i \leq o+1$ $\wedge(\forall_{1\leq l<i})(l\%2 = 1 \longrightarrow a[l] \leq 0) \wedge (o \leq N \longrightarrow a[o] > 0)$
p_1'	$\triangleq$	$1 \leq o' \leq N+1 \wedge i'\%2 = 1 \wedge 1 \leq i' \leq o'+1$ $\wedge(\forall_{1\leq l<i'})(l\%2 = 1 \longrightarrow a[l] \leq 0) \wedge (o' \leq N \longrightarrow a[o'] > 0)$
q_1	$\triangleq$	$1 \leq o' \leq N+1 \wedge (\forall_{1\leq l<o'})(l\%2 = 1 \longrightarrow a[l] \leq 0) \wedge (o' \leq N \longrightarrow a[o'] > 0)$
p_2	$\triangleq$	$2 \leq e \leq N+1 \wedge j\%2 = 0 \wedge 2 \leq j \leq e+1$ $\wedge(\forall_{1\leq l<j})(l\%2 = 0 \longrightarrow a[l] \leq 0) \wedge (e \leq N \longrightarrow a[e] > 0)$
p_2'	$\triangleq$	$2 \leq e' \leq N+1 \wedge j'\%2 = 0 \wedge 2 \leq j' \leq e'+1$ $\wedge(\forall_{1\leq l<j'})(l\%2 = 0 \longrightarrow a[l] \leq 0) \wedge (e' \leq N \longrightarrow a[e'] > 0)$
q_2	$\triangleq$	$2 \leq e' \leq N+1 \wedge (\forall_{1\leq l<e'})(l\%2 = 0 \longrightarrow a[l] \leq 0) \wedge (e' \leq N \longrightarrow a[e'] > 0)$
$post$	$\triangleq$	$1 \leq k' \leq N+1 \wedge (\forall_{1\leq l<k'})(a[l] \leq 0) \wedge (k' \leq N \longrightarrow a[k'] > 0)$

Map macros for environment and store		
Env	$\triangleq$	$\mathtt{a} \mapsto \mathbf{a}\ \mathtt{i} \mapsto \mathbf{i}\ \mathtt{j} \mapsto \mathbf{j}\ \mathtt{oddtop} \mapsto \mathbf{o}\ \mathtt{eventop} \mapsto \mathbf{e}\ \mathtt{N} \mapsto \mathbf{N}\ \mathtt{k} \mapsto \mathbf{k}$
St	$\triangleq$	$\mathbf{a} \mapsto a\ \mathbf{i} \mapsto i\ \mathbf{j} \mapsto j\ \mathbf{o} \mapsto o\ \mathbf{e} \mapsto e\ \mathbf{N} \mapsto N\ \mathbf{k} \mapsto k$
St'	$\triangleq$	$\mathbf{a} \mapsto a\ \mathbf{i} \mapsto i'\ \mathbf{j} \mapsto j'\ \mathbf{o} \mapsto o'\ \mathbf{e} \mapsto e'\ \mathbf{N} \mapsto N\ \mathbf{k} \mapsto k'$

RL formulas			
($\clubsuit$)	$\langle\langle \mathtt{S1} \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St \rangle_{\mathsf{st}} \wedge i < o \wedge p_1$	$\Rightarrow$	$\langle\langle \cdot \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St' \rangle_{\mathsf{st}} \wedge o' \leq i' \wedge p_1' \wedge q_1$
($\diamondsuit$)	$\langle\langle \mathtt{S2} \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St \rangle_{\mathsf{st}} \wedge j < e \wedge p_2$	$\Rightarrow$	$\langle\langle \cdot \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St' \rangle_{\mathsf{st}} \wedge e' \leq j' \wedge p_2' \wedge q_2$
($\spadesuit$)	$\langle\langle \mathtt{FIND} \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St \rangle_{\mathsf{st}} \wedge pre$	$\Rightarrow$	$\langle\langle \cdot \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St' \rangle_{\mathsf{st}} \wedge post$

Corresponding proofs given by `kcheck`
$s(i) \triangleq$ [CaseAnalysis], ([SymbolicStep]) $\vee$ ([SymbolicStep]),[CircularHypothesis](i)
($\clubsuit$) [SymbolicStep], [CaseAnalysis], [Implication] $\vee$ ($s(\clubsuit)$, [Implication])
($\diamondsuit$) [SymbolicStep], [CaseAnalysis], [Implication] $\vee$ ($s(\diamondsuit)$, [Implication])
($\spadesuit$) [SymbolicStep] $\times 5$, [CircularHypothesis](1), [CircularHypothesis](2), [Implication]

Figure 7: RL formulas necessary to verify `FIND`. We use `a`, `i`, `j`, `oddtop`, `eventop`, `N`, `k` to denote program variables, $\mathbf{a}$, $\mathbf{i}$, $\mathbf{j}$, $\mathbf{o}$, $\mathbf{e}$, $\mathbf{N}$, $\mathbf{k}$ to denote locations, and a, i, j, o, e, N, k for variables values. We also use $s(i)$ to denote a common sequence in the proofs of ($\clubsuit$) and ($\diamondsuit$). CaseAnalysis splits the proof in two goals separated by $\vee$, while CircularHypothesis(i) represents the application of the formula (i) as a circularity. [SymbolicStep] $\times n$ is the equivalent of applying [SymbolicStep] n times.



RESEARCH CENTRE
LILLE – NORD EUROPE

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

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

ISSN 0249-6399