Language-Independent Program Verification Using Symbolic Execution

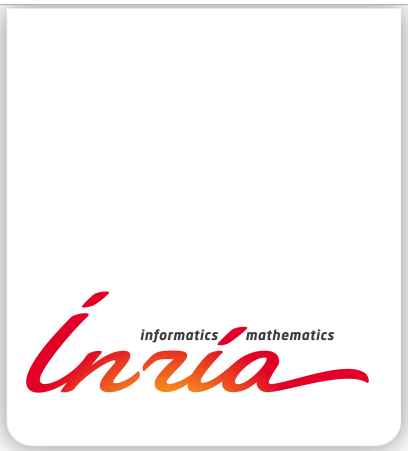
Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

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 — 21 pages

Abstract: We present an automatic, language-independent program verification approach and experimental 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, which offers a lot of freedom (but very few guidelines) for constructing proofs. Hence, we propose in this paper an alternative proof system, in which symbolic execution becomes a rule for proof construction. We show that under reasonable conditions on the semantics of programming languages our proof system is sound. We then present a Reachability-Logic verifier based on our proof system, which is implemented in the $\mathbb{K}$ framework and illustrated on programs written in languages also 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 a un système déductif qui est correct et relativement complet, qui laisse beaucoup de liberté à l'utilisateur sur la manière dâappliquer les règles de déduction, mais qui nâoffre pas de mode d'emploi pour construire des preuves. Nous montrons que l'on peut utiliser une méthode générique d'exécution symbolique de programmes, que nous avons introduite récemment, comme une stratégie de construction de preuves dans ce système déductif. Nous montrons que, moyennant des conditions raisonnables sur la sémantique des langages de programmation, notre méthode de vérification est correcte. Nous présentons une implémentation prototype d'un outil de vérification basé sur ces idées, que nous avons implémenté dans la K framework et que nous illustrons sur des exemples de programmes écrits dans des langages formellement définis en K.

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

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

Figure 1: Program `gcd`

1 Introduction

Reachability Logic (RL) [22] is a language-independent logic for specifying program properties. For instance, on the `gcd` program in Fig. 1, the RL formula

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

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-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 attempt does not succeed because its gets into an infinite symbolic execution, induced by the infinitely many iterations of the loop. Although symbolic execution alone is not enough for formal verification, it is useful for testing, since one symbolic execution covers (possibly, infinitely) many test cases. For example, on the `gcd` program in Figure 1, one can test by a finite symbolic execution that the program works correctly for all $a, b \geq 0$ such that $a = k \times b$ for any $k \in \mathbb{N}$.

Reachability Logic's proof system [22] is a set of seven inference rules that is sound and relatively complete, meaning that it proves exactly all valid reachability formulas. It is a compact and elegant proof system but, despite its nice theoretical properties, its use in practice for proving nontrivial programs is difficult, because it gives the user a lot of freedom regarding the order and manner of rule application, and offers no practical guidelines for constructing proofs. It appears to have been designed for obtaining the soundness and relative completeness meta-theoretical results, but less so for actually verifying programs.

Contribution

A language-independent approach and experimental tool for proving properties of programs expressed in RL. The approach consists in an alternative proof system, where symbolic execution is a main ingredient and is used as a rule during proof construction. The tool executes our proof system with a certain strategy, giving priority to re-using proof goals as hypotheses. The soundness of the proposed approach is also based on a so-called *circularity principle* for reachability-logic formulas, which specifies the conditions under which RL goals can be reused as hypotheses in proofs of programs. This is essential for proving programs with infinite state-spaces induced e.g., by performing an unbounded (symbolic) number of loop iterations or of recursive calls. Soundness moreover requires that the semantics of the programming language be *total* (or, at least,

that it can be transformed into an total semantics in a simple way). We implemented the verifier as a prototype tool in the $\mathbb{K}$ framework [24]. The tool is illustrated on some programs written in languages also formally defined in $\mathbb{K}$.

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 [22]. Section 3 contains the core contribution of the paper. We first present the main ingredients of a novel generic approach for symbolic execution, which appeared in an prelminary form earlier in [6]. We then introduce an alternative, symbolic-execution-based proof system for verifying RL formulas, whose soundness is based on a circularity principle for RL. Section 4 describes an experimental verification tool based on our language-independent symbolic execution tool [6] and its application to a parallel program written in a language defined in $\mathbb{K}$. The paper ends with a description of related work. An appendix (for reviewers) contains the proofs of the technical results in the paper. Our tool (with instructions) 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 partially supported by Contract 161/15.06.2010, SMISC-SNR 602-12516 (DAK) and 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. 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 a set Π of predicate symbols. Σ includes at least a sort *Cfg* for *configurations* as well as sorts for the syntax of the language $\mathcal{L}$, e.g., expressions and statements. Σ may also include other data sorts, depending on the datatypes occuring in the language $\mathcal{L}$ (e.g., Booleans, integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all *data* sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{\mathsf{Data}}$. Let T_Σ denote the Σ-algebra of ground terms and $T_{\Sigma,s}$ denote the set of ground terms of sort s. Given a sort-wise set of variables V, let $T_\Sigma(V)$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(V)$ denote the set of terms of sort s with variables, and $var(t)$ denote the set of variables occurring in the term t.

Model:

$\mathcal{T}$ is a Φ-model, i.e., it interprets every function and predicate in Φ. We assume that it interprets the data sorts and their operations according to a given Σ^{Data}-model $\mathcal{D}$. For simplicity, we write in the sequel $true, false, 0, 1, \ldots$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}, \mathcal{D}_0, \mathcal{D}_1$, etc. $\mathcal{T}$ interprets the non-data

sorts as the free Σ-model generated by $\mathcal{D}$, i.e., as ground terms over the signature $(\Sigma \setminus \Sigma^{\mathsf{Data}}) \cup \mathcal{D}$. We denote by $\rho \models \phi$ the satisfaction of a Φ-formulas ϕ by a valuation $\rho : \mathit{Var} \to \mathcal{T}$.

Rules:

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

Definition 1 (pattern over a given set of variables [22]) *An* elementary pattern *over a set of variables Var is an expression of the form $\pi \wedge \phi$, where $\pi \in T_{\Sigma, \mathit{Cfg}}(\mathit{Var})$ is a* basic pattern *and ϕ is a Φ-formula called the pattern's* condition. *If $\gamma \in T_{\mathit{Cfg}}$ and $\rho : \mathit{Var} \to \mathcal{T}$ we write $(\gamma, \rho) \models \pi \wedge \phi$ for $\gamma = \pi\rho$ and $\rho \models \phi$. We let $[\![\varphi]\!]$ denote the set $\{\gamma \in \mathcal{T}_{\mathit{Cfg}} \mid$ there is $\rho : \mathit{Var} \to \mathcal{T}$ s.t. $(\gamma, \rho) \models \varphi\}$.*

The above definition is a particular case of a definition in [22]. There, a pattern is a first-order logic formula with configuration basic-patterns as sub-formulas.

A basic pattern π defines a set of (concrete) configurations, and the condition ϕ gives additional constraints these configurations must satisfy. We identify basic patterns π with elementary patterns $\pi \wedge \mathit{true}$.

Sample patterns are $\langle\langle I_1 + I_2 \curvearrowright C\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}} \langle \mathit{Env}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{\mathit{Int}} 0$.

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

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-explained since it uses a BNF notation. The statements of the language are either assignments, *if* statements, *while* loops, *skip* (i.e., the empty statement), blocks 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 natural numbers then it only concerns the arguments whose positions are present in the list. The operational semantics of IMP is given as (possibly conditional) rewrite rules over *configurations*. Configurations typically contain the program to be executed, together with any additional information required for program execution. The structure configurations depends on the language being defined; for IMP, it consists only of the code to be executed and an environment mapping variables to values:

$$\mathit{Cfg} \quad ::= \langle\langle \mathit{Code}\rangle_{\mathsf{k}} \langle \mathit{Map}_{\mathit{Id},\mathit{Int}}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$$

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:

$$
\begin{array}{rll}
\multicolumn{3}{l}{Id ::= \text{domain of identifiers}} \\
\multicolumn{3}{l}{Int ::= \text{domain of integer numbers}} \\
\multicolumn{3}{l}{Bool ::= \text{domain of boolean constants}} \\
AExp ::= Int & & \mid AExp \texttt{ / } AExp \,[\text{strict}] \\
\mid Id & & \mid AExp \texttt{ * } AExp \,[\text{strict}] \\
\mid (AExp) & & \mid AExp \texttt{ + } AExp \,[\text{strict}] \\
\mid AExp \texttt{ / } AExp \,[\text{strict}] & & \mid AExp \texttt{ \% } AExp \,[\text{strict}] \\
BExp ::= Bool \mid (BExp) & & \\
\quad \mid AExp \texttt{ <= } AExp \,[\text{strict}] & & \\
\quad \mid \texttt{not } BExp \,[\text{strict}] & & \\
\quad \mid BExp \texttt{ and } BExp \,[\text{strict}(1)] & & \\
Stmt ::= \texttt{\{\}} \mid \texttt{\{ } Stmt \texttt{ \}} \mid Stmt \texttt{ ; } Stmt & & \\
\quad \mid Id \texttt{ := } AExp, [\text{strict}(2)] & & \\
\quad \mid \texttt{while } BExp \texttt{ do } Stmt & & \\
\quad \mid \texttt{if } BExp \texttt{ then } Stmt \texttt{ else } Stmt \,[\text{strict}(1)] & & \\
Code ::= AExp \mid BExp \mid Stmt \mid Code \curvearrowright Code & &
\end{array}
$$

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

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

Here, BE ranges over $BExp \setminus \{false, true\}$, B ranges over the Boolean values $\{false, true\}$, and $\square$ is a special variable, destined to receive the value of BE once it is computed, typically, by applying the other rules in the semantics.

We show how the definition of IMP fits the theoretical framework given in Section 2.1. Nonterminals from the syntax (Int, $Bool$, $AExp$, ...) are sorts in Σ. Each production from the syntax defines an operation in Σ; e.g, the production $AExp ::= AExp \texttt{ + } AExp$ defines the operation $\_+\_ : AExp \times AExp \to AExp$. These operations define the constructors of the result sort. For the sort Cfg, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : Code \times Map_{Id,Int} \to Cfg$. The expression $\langle\langle I_1 \texttt{ / } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$ is an elementary pattern in which $=_{Int}$ is a predicate symbol, I_1, I_2 are variable of sort Int, C is a variable of sort $Code$ (the rest of the computation), and Env is a variable of sort $Map_{Id,Int}$ (the rest of the environment). The data algebra $\mathcal{D}$ interprets Int as the set of integers, the operations like $+_{Int}$ (cf. Figure 3) as the corresponding usual operation on integers, $Bool$ as the set of Boolean values $\{false, true\}$, the operation like $\wedge$ as the usual Boolean operations, the sort $Map_{Id,Int}$ as the multiset of maps $X \mapsto I$, where X ranges over identifiers Id and I over the integers. Predicate symbols such as $=_{Int}, \leq_{Int}$ are interpreted by the corresponding predicates over integers. The value of an identifier X is an environment M is $lookup(X, M)$, and the environment M, updated by binding an identifier X to a value I, is $update(X, M, I)$. Here, $lookup()$ and $update()$ are operations in a signature $\Sigma^{\mathsf{Map}} \subseteq \Sigma^{\mathsf{Data}}$ of maps. The other sorts, $AExp$, $BExp$, $Stmt$, and $Code$, are interpreted in the algebra $\mathcal{T}$ as ground terms in which data subterms are replaced by their interpretations, e.g., $\texttt{if } 1 >_{Int} 0 \texttt{ then \{\} else \{\}}$ is intepreted as $\texttt{if } \mathcal{D}_{true} \texttt{ then \{\} else \{\}}$.

2.3 Reachability Logic

We recall the syntax and semantics of a subset of reachability logic (RL) [22]. The formulas we consider are the most commonly used ones for defining a language's semantics and for stating properties about programs. Then the RL proof system togetheter with its soundness property [22]

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

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

are briefly presented; these are essential in showing the correctness of our symbolic execution-based verification.

We consider the subset of reachability logic formulas that are pairs $\varphi_1 \Rightarrow \varphi_2$, where the left and right-hand sides are disjunctions of patterns (cf. Definition 1) over a set of variables, say, *Var*. In particular, all semantical rules of a language are of this kind (cf. Definition 2). RL formulas can be more general, as they may have conditions that are themselves RL formulas. However, we found the considered subset expressive enough for defining a language's semantics and for stating properties about programs. For simplicity we still call this subset RL.

Semantically, a disjunction $\varphi \triangleq \bigvee_{i\in I} \varphi_i$ of patterns is satisfied by a configuration γ and a valuation ρ, written $(\gamma, \rho) \models \varphi$, if $(\gamma, \rho) \models \varphi_i$ for some $i \in I$. 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 $\gamma_1 \overset{*}{\Longrightarrow}_{\mathcal{S}} \gamma_2$ in $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$.
We consider here the version of the reachability logic proof system described in [22]. It 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 $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$. The validity of a FOL formula f is denoted $\models f$, and the left and right-hand patterns of reachability-logic formulas are interpreted as FOL formulas [22]. $\varphi \wedge \phi$, for $\varphi \triangleq \bigvee_{i\in I} \pi_i \wedge \phi_i$, is a shortcut for the formula $\bigvee_{i\in I} \pi_i \wedge (\phi_i \wedge \phi)$. RL formulas may, in general, include quantifiers (as in the [Abstraction] rule) but here we will be using it only for unquantified patterns. 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 system In Figure 4 is *sound* [22]: 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 *relative completeness* result, not used here.

The proof system in Figure 4 leaves a lot of freedom to the user as to which rule to apply when verifying programs. At any step of the proof, one must choose, whether to include the current goal in the circularities G, i.e., to apply the Circularity rule; or to derive some intermediate goal φ'' using the Transitivity rule, and to continue from there on by using the current set of circularities as new semantical rules; or to split the premise of the current goal into two formulas using

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

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

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

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

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

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

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

Figure 4: Proof System for RL.

the CaseAnalysis rule, which, thanks to the Consequence rule, can be arbitrary, provided their disjunction is logically equivalent to the premise. This freedom (and lack of guidance) makes it difficult to use this proof system when actually verifying programs. In the following sections propose another approach, based on another proof system, which is more "operational" than the original one.

3 Symbolic Execution for Reachability-Logic Verification

Symbolic execution consists in executing programs with symbolic values instead of concrete ones. We briefly present a novel approach to language-independent symbolic execution, a preliminary version of which appeared earlier in [6]. We then show how symbolic execution can form the basis of formal verification.

3.1 Symbolic Execution

Consider a language definition $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$. In order to symbolically execute programs in $\mathcal{L}$, it is enough to consider, instead of the model $\mathcal{T}_{Cfg}$, the set of patterns of the form $\pi \wedge \phi$, with π a term in the $(\Sigma \setminus \Sigma^{\mathsf{Data}}) \cup \mathcal{D}$-algebra of terms of sort *Cfg*, and ϕ a Φ-formula, with free variables over a set *Var*; and to apply the rules in $\mathcal{S}$ with *unification*, instead of matching as required by Definition 2.

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

Note that we called a symbolic unifier in the above definition a *most general unifier*, even though the standard notion of most general unifier in rewriting is a different one. Applying rules with unification uses the following observation.

Remark 1 *Any pattern $\pi \wedge \phi$ can be transformed into another pattern $\pi' \wedge \phi'$ satisfying $[\![\pi \wedge \phi]\!] = [\![\pi' \wedge \phi']\!]$, and such that π' is linear (i.e. no variable occurs twice) and all its data subterms are variables. For this, just replace all duplicated variables and all non-variable data subterms in π by fresh variables, and add constraints to equate in ϕ these variables to the subterms they replaced.*

Example 1 *The pattern $\langle\langle X \;/\; Y\rangle_{\mathsf{k}}\langle Y \mapsto A +_{Int} 1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge A \neq_{Int} -1$ with X, Y variables of sort Id and A of sort Int is nonlinear because Y occurs twice. It contains the non-variable data term $A +_{Int} 1$. It is transformed into*

$$\langle\langle X \;/\; Y\rangle_{\mathsf{k}}\langle Y' \mapsto A'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge Y' =_{Id} Y \wedge_{Bool} A' =_{Int} A +_{Int} 1 \wedge_{Bool} A \neq_{Int} -1$$

.

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

Lemma 1 (Unification by Matching) *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 $\sigma : var(t_1) \mapsto T_\Sigma(var(t_2))$ such that $t_1\sigma = t_2$ and such that $\sigma_{t_2}^{t_1} \triangleq \sigma \uplus id_{var(t_2)}$ is a most-general unifier of t_1, t_2.*

The most general unifier $\sigma_{t_2}^{t_1}$ is unique since is defined to be $\sigma \uplus id_{var(t_2)}$ and σ, which is a (syntactical) match of t_1 on t_2, is unique when it exists.

We now define the symbolic transition relation. For patterns $\varphi = \pi \wedge \phi, \varphi' = \pi' \wedge \phi'$, we let $\varphi \sim \varphi'$ iff $t' = t$ and the equivalence $\phi' \leftrightarrow \phi$ is logically valid. $\sim$ is an equivalence relation; we denote by $[\varphi]_\sim$ the equivalence class of φ w.r.t. $\sim$.

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

Remark 2 *The symbolic transition relation is finitely branching: every $[\varphi]_\sim$ has finitely many successors since there are at most finitely many rules in $\mathcal{S}$ that match the basic pattern of φ, and the (possibly, infinitely many) patterns equivalent to the those generated by the rules are collapsed into an equivalence class.*

We prove in [5] that the concrete transition relation $\Rightarrow_{\mathcal{S}}$ and the restriction of the symbolic transition $\Rightarrow_{\mathcal{S}}^{\mathfrak{s}}$ to *satisfiable* patterns mutually simulate each other. This ensures, in particular, that it is sound to use symbolic execution in order to perform program testing, which is more efficient than concrete execution because one symbolic execution includes possibly infinitely many concrete test cases.

Assumption 1 *Hereafter we assume that for all elementary patterns $\varphi \triangleq \pi \wedge \phi$, $\varphi' \triangleq \pi' \wedge \phi'$ such that $[\varphi]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi']_\sim$, π and π' may only have variable of data sorts. This can be obtained by starting with an initial pattern satisfying these properties and by ensuring that these properties are preserved by the rules in $\mathcal{S}$. That is, in our symbolic-execution framework, only the data may be symbolic.*

3.2 Reachability-Logic Formula Verification

We show in this section how symbolic execution can be included in a proof system for RL formulas. We first define, using the symbolic transition relation in Definition 4, the essential concept of *derivative* that occurs in our proof system. It uses the choice operation ε, which picks an arbitrary element in a nonempty set.

Definition 5 *The* derivative $\Delta_{\mathcal{S}}(\varphi)$ *of a pattern* φ *for a set* $\mathcal{S}$ *of rules is*

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

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

Remark 3 *Since the symbolic transition relation is finitely branching (Remark 2). for finite rule sets* $\mathcal{S}$ *the derivative is a finite disjunction. Note also that the patterns in the derivative are only defined up to the equivalence relation* $\sim$*.*

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

Definition 6 (Total Semantics) *We say that a set* $\mathcal{S}$ *of semantical rules is* total *if for each basic pattern* π_1 *occurring in the left-hand side of a rule, the disjunction* $\bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} (\phi_1 \wedge \phi_2)$ *is valid in* $\mathcal{T}$*.*

Remark 4 *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 6 because the "disjunction" in that definition reduces to* $I_2 \neq 0$*, which is not 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.*

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 7 (Cover) *Consider an elementary pattern* $\varphi \triangleq \pi \wedge \phi$*. A set of rules* $\mathcal{S}'$ *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 *The existence of the most-general unifier* $\sigma_\pi^{\pi_1}$ *in the above definition means the basic patterns in the LHS of rules in* $\mathcal{S}'$ *are unifiable with the basic pattern* π*. In particular,* $\bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} (\phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$ *is a nonempty disjunction, otherwise the validity in Def. 7 would not hold (an empty disjunction is false).*

Lemma 2 *If* $\mathcal{S}$ *is total and* φ *is derivable for* $\mathcal{S}$ *then* $\mathcal{S}$ *is a cover for* φ*.*

Using the notion of cover we obtain a derived rule of the RL proof system:

Lemma 3 *If* $\mathcal{S}' \subseteq \mathcal{S}$ *is a cover for* φ*, and* G *is a (possibly empty) set of RL formulas, then* $\mathcal{S} \vdash_G \varphi \Rightarrow \Delta_{\mathcal{S}'}(\varphi)$*.*

Corollary 1 *If* $\mathcal{S}$ *is total and* φ *is derivable for* $\mathcal{S}$*, then* $\mathcal{S} \vdash \varphi \Rightarrow \Delta_{\mathcal{S}}(\varphi)$*.*

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

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

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

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

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

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

Before we introduce our proof system we need to deal with the issue that operational semantics are not always weakly well-defined as required by the RL original deductive system's soundness. For example, the semantics of IMP is not weakly well-defined due to the rules for division and modulo, which, for valuations ρ mapping divisors to 0, have no instance of their right-hand side. 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 4), 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 weakly well-defined. This transformation is formalised as follows.

Definition 8 ($\mathcal{S}^{\Delta}$) *Given a set of semantical rules $\mathcal{S}$, the set of semantical rules $\mathcal{S}^{\Delta}$ is defined by $\mathcal{S}^{\Delta} \triangleq \{\pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \mid (\pi \wedge \phi \Rightarrow \varphi) \in \mathcal{S}\}$.*

Definition 9 (Weakly Well-Definable Semantics) *We say that a set of semantical rules $\mathcal{S}$ is* weakly well-definable *if $\mathcal{S}^{\Delta}$ is weakly well-defined.*

If a semantics $\mathcal{S}$ is not weakly well-defined, but only weaky well-definable (which happens quite often - e.g., all the languages defined in the $\mathbb{K}$ framework that have rules for numeric division are in this case) then one can use the $\mathcal{S}^{\Delta}$ semantics, which is by definition weakly well-defined. We note that in this case $\mathcal{S}^{\Delta}$ is also (trivially) total, which is required by the soundness of our approach.

A few other properties useful in the sequel are stated and proved below.

Lemma 4 *$\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff $\mathcal{S}^{\Delta} \models \varphi \Rightarrow \varphi'$.*

Lemma 5 *A pattern φ is derivable for $\mathcal{S}$ iff φ is derivable for $\mathcal{S}^{\Delta}$.*

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 $\{\Delta_{\mathcal{S}}(\varphi) \Rightarrow \varphi' \mid \varphi \Rightarrow \varphi' \in G\}$, $\mathcal{S} \Vdash G$ denotes the conjunction $\bigwedge_{\varphi \Rightarrow \varphi' \in G} \mathcal{S} \Vdash \varphi \Rightarrow \varphi'$, and $\mathcal{S} \models G$ denotes $\bigwedge_{\varphi \Rightarrow \varphi' \in G} \mathcal{S} \models \varphi \Rightarrow \ \varphi'$. The proof system $\Vdash$ is shown in Figure 5

Theorem 1 (Circularity Principle for RL) *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 $\mathcal{S} \models G$.*

If a semantics $\mathcal{S}$ is not weakly well-defined but only weakly-well definable, one can use Theorem 1 with $\mathcal{S}^{\Delta}$ instead of $\mathcal{S}$, and, under the same hypotheses for the derivability of G, one can deduce $\mathcal{S} \models G$, thanks to Lemmas 4 and 5.

Example 2 *We show how the RL formula (1) is proved using our $\Vdash$ deductive system, which amounts to verifying that the* `gcd` *program meets its specification. For this, we consider the two following formulas, where* `while` *denotes the program fragment consisting of the* `while` *loop, and* `body` *denotes the loop's body:*

$$\langle\langle \texttt{while} \rangle_{\mathsf{k}} \langle a \mapsto a \ b \mapsto b \ x \mapsto x \ y \mapsto y \ r \mapsto r \rangle_{\mathsf{env}} \rangle_{\mathrm{cfg}} \wedge gcd(a,b) = gcd(x,y) \wedge x \geq 0 \wedge y \geq 0 \Rightarrow$$
$$\langle\langle \cdot \rangle_{\mathsf{k}} \langle a \mapsto a \ b \mapsto b \ x \mapsto x' \ y \mapsto y' \ r \mapsto r' \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge gcd(a,b) = gcd(x',y') \wedge x' \geq 0 \wedge y' = 0 \quad (2)$$
$$\langle\langle \texttt{body} \cdots \rangle_{\mathsf{k}} \langle a \mapsto a \ b \mapsto b \ x \mapsto x \ y \mapsto y \ r \mapsto r \rangle_{\mathsf{env}} \rangle_{\mathrm{cfg}} \wedge gcd(a,b) = gcd(x,y) \wedge x > 0 \wedge y \geq 0 \Rightarrow$$
$$\langle\langle \cdot \rangle_{\mathsf{k}} \langle a \mapsto a \ b \mapsto b \ x \mapsto x' \ y \mapsto y' \ r \mapsto r' \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge gcd(a,b) = gcd(x',y') \wedge x' \geq 0 \wedge y' \geq 0 \quad (3)$$

The rules say that the `while` *loop and its body preserve an invariant: the gcd of the values of* `a`*,* `b` *equals the gcd of the values of* `x`*,* `y`*. There are pre and post-conditions on these values, with slight differences between the two rules.*

We consider the set of goals G consisting of the formulas (1), (2), and (3), and apply the deductive system $\Vdash$ to these goals in order to prove them:

- *the formula (1) is proved by applying a number of times the SymbolicStep rule until the* `k` *cell contains the* `while` *program fragment. Then, the CircularHypothesis rule is applied with the formula (2). Next, the Implication rule is used to prove that the pattern resulting from applying the formula (2) implies the right-hand side of the formula (1). Finally, the Transitivity rule builds a proof of (1) from the individual rule applications described above.*

- *the formula (2) is proved by applying a number of times the SymbolicStep and CaseAnalysis rules, until the program remaining to be executed is either:*

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

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

4 An Experimental Tool

In this section we describe the implementation of a tool `kcheck`, which uses the $\Vdash$ deductive system that we have defined in order to verify $\mathcal{S} \models G$, for a given language semantics $\mathcal{S}$ and a set of reachability formulas (goals) G. The implementation is part of the $\mathbb{K}$ tools suite [2] and it has been developed on top of our symbolic execution tool [6]. $\mathbb{K}$ 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 [19], C [14], and Java [8]. An example of a $\mathbb{K}$ definition can be found in Section 2.2. The framework also includes support for symbolic execution, based on the earlier approach [6], which is different in terms of formalism, yet equivalent to the one introduced here in Section 3.1. In terms of implementation, our `kcheck` prototype reuses components of the $\mathbb{K}$ framework: parsing, compilation steps, support for symbolic execution, and connections to Maude's [18] state-space exlporer and to the Z3 SMT solver [13].

The tool first builds a new definition and then performs verification. Given a language definition $\mathcal{S}$ and a set of RL formulas G, `kcheck` produces a new definition $\mathcal{S}^{\mathfrak{s}} \cup G^{\mathfrak{s}}$, where $^{\mathfrak{s}}$ is a rule transformation such that standard rewriting with them implements symbolic rewriting [6]. If $\mathcal{S}$ is weakly well-definable (Def. 9) but not weakly well-defined then $\mathcal{S}^{\Delta}$ is used as a starting point instead of $\mathcal{S}$. This new definition is used to perform symbolic execution of the patterns in left hand sides of formulas in $\Delta_{\mathcal{S}}(G)$, in order to find a proof of $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ with the $\Vdash$ proof system that we have defined.

Computing all the successors of a pattern by symbolically applying rules from $\mathcal{S}$ corresponds to applying the SymbolicStep deduction rule. Computing the successor of a pattern by using a rule from G corresponds to applying CircularHypothesis. The rules from G are only applied to patterns from the left hand side of $\Delta_{\mathcal{S}}(G)$ or to their successors obtained by derivation. This is achieved using a labeling mechanism that singles out the patterns to which rules from G can be applied, and, moreover, this gives priority to rules in G over rules in $\mathcal{S}$. CaseAnalysis is implicitly applied by splitting disjunctive patterns $\bigvee_i(\pi_i \wedge \phi_i)$ obtained by SymbolicStep and CircularHypothesis back into elementary patterns $\pi_i \wedge \phi_i$. Implication is applied at the end of the branches of the proof tree, in order to check that the current pattern implies the right hand side of the current goal. This has to succeed in all branches for the proof to succeed. Finally, Transitivity builds a full proof in the $\Vdash$ deductive system from individual proof steps.

We have used `kcheck` to prove `gcd.imp` (Figure 1) as sketched in Example 2. 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 other $\mathbb{K}$ language definitions as well, as demonstrated in the following example.

4.0.1 Verifying a parallel program: `FIND`

The program is borrowed 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.

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`. To facilitate the semantics of threads, more specifically to naturally give them 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

```
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.

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:

$$\langle\langle\langle Code\rangle_{\mathsf{k}}\langle Map_{Id,Int}\rangle_{\mathsf{env}}\langle Int\rangle_{\mathsf{id}}\rangle_{\mathsf{th}^+}\langle Map_{Int,Int}\rangle_{\mathsf{st}}\rangle_{\mathsf{cfg}}$$

Most of the syntactical constructs of this language have almost the same semantics as in IMP (e.g. assignments, if-statements, loops). However, the language is more complex than IMP, since it supports arrays and threads. The main thread, identified by 0, creates a new $\langle\rangle_{\mathsf{th}}$ cell, each time when the operator `||` is met. For instance, when `S1||S2;` has to be executed, the main thread creates $\langle\langle S1\rangle_{\mathsf{k}}\langle\rho\rangle_{\mathsf{env}}\langle id_1\rangle_{\mathsf{id}}\rangle_{\mathsf{th}}$ and $\langle\langle S2\rangle_{\mathsf{k}}\langle\rho\rangle_{\mathsf{env}}\langle id_2\rangle_{\mathsf{id}}\rangle_{\mathsf{th}}$, where ρ is the map of variables inherited from the main thread, and id_1 and id_2 are two fresh identifiers. Thread 0 waits for these two threads to finish and then computes the return value k.

The `||` operator yields a non-deterministic behavior of `FIND`. However, in [4] the authors prove that all computations of a disjoint parallel program starting in the same initial state produce the same output. For program verification this observation simplifies matters because it allows independent verification of the parallel code, without considering the interleavings caused by parallelism.

Figure 7 shows all the ingredients that we used to prove `FIND` using our tool. At the figure's top we show the code macros that we use in our reachability 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. Next, we show the RL formulas (1-8), corresponding to the set G in our $\Vdash$ deduction system, which were constructed from the above pre/post conditions and invariants. Finally we show the proofs automatically constructed by `kcheck`.

The proof tree corresponding to sequential of code (e.g. `INIT`) has a single branch, while the proof tree corresponding to code containing loops or if-statements can have multiple branches. Some examples are the proofs for the RL formulas (3), (5), and (7), where CaseAnalysis splits the proof in two branches. The proof tree for the RL formula (8), corresponding to the specification of `FIND`, has a single branch because it uses circularities that do not split the proof tree.

The verification of `FIND` might look difficult, but this is because its verification is hard in general, no matter what underlying logic one uses. Our proof has the same structures as the proof from [4], except for the fact that we express program properties as RL formulas. However, when performing mechanised verification, the pre/post conditions and the invariants must be very accurate. Otherwise, the proof will fail even if, intuitively, all the formulas seem to hols. For example, 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 ≥ 2, a constraint forgotten in [4].

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	$\triangleq$	$2 \leq e' \leq N+1 \wedge (\forall_{1\leq l<e'})(l\%2 = 0 \longrightarrow a[l] \leq 0) \wedge (e' \leq N \longrightarrow a[e'] > 0)$
m	$\triangleq$	$1 \leq min(o,e) \leq N+1 \wedge (\forall_{1\leq l<min(o,e)})(a[l] \leq 0)$
		$\wedge(min(o,e) \leq N \longrightarrow a[min(o,e)] > 0)$
$post$	$\triangleq$	$1 \leq k' \leq N+1 \wedge (\forall_{1\leq l<k'})(a[l] \leq 0) \wedge (k' \leq N \longrightarrow a[k'] > 0)$

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'	$\triangleq$	$\mathbf{a} \mapsto a\ \mathbf{i} \mapsto i'\ \mathbf{j} \mapsto j'\ \mathbf{o} \mapsto o'\ \mathbf{e} \mapsto e'\ \mathbf{N} \mapsto N\ \mathbf{k} \mapsto k'$

Reachability rules

(1)	$\langle\langle \mathtt{INIT}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge pre$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge (p_1' \wedge p_2')$
(2)	$\langle\langle \mathtt{BODY1}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge i < o \wedge p_1$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge p_1'$
(3)	$\langle\langle \mathtt{S1}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge i < o \wedge p_1$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge o' \leq i' \wedge p_1' \wedge q_1$
(4)	$\langle\langle \mathtt{BODY2}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge j < e \wedge p_2$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge p_2'$
(5)	$\langle\langle \mathtt{S2}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge j < e \wedge p_2$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge e' \leq j' \wedge p_2' \wedge q_2$
(6)	$\langle\langle \mathtt{S1\|\|S2;}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge p_1 \wedge p_2$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge q_1 \wedge q_2$
(7)	$\langle\langle \mathtt{MIN}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge q_1 \wedge q_2 \wedge m$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge post$
(8)	$\langle\langle \mathtt{FIND}\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge pre$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge post$

Corresponding proofs given by `kcheck`

(1) [SymbolicStep], [SymbolicStep], [SymbolicStep], [SymbolicStep], [Implication]
(2) [CaseAnalysis], ([SymbolicStep], [Implication]) $\vee$ ([SymbolicStep], [Implication])
(3) [SymbolicStep], [CaseAnalysis], ([Implication]$\vee$(2),(3), [Implication])
(4) [CaseAnalysis], ([SymbolicStep], [Implication]) $\vee$ ([SymbolicStep], [Implication])
(5) [SymbolicStep], [CaseAnalysis], ([Implication]$\vee$(4),(5), [Implication])
(6) [SymbolicStep],(3),(5), [Implication]
(7) [CaseAnalysis], ([SymbolicStep], [Implication]) $\vee$ ([SymbolicStep], [Implication])
(8) (1),(6),(7), [Implication]

Figure 7: RL formulas necessary to verify `FIND`. We use `a`, `i`, `j`, `oddtop`, `eventop`, `N`, `k` to denote program variables, **a**, **i**, **j**, **o**, **e**, **N**, **k** to denote locations, and a, i, j, o, e, N, k for variables values. By $min(o,e)$ we refer the mathematical function that returns the minimum of o and e. CaseAnalysis splits the proof in two goals separated by $\vee$, while the application of CircularHypothesis where α is the i-the formula is represented as (i).

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.

5 Conclusion, Related Work, and Future Work

We have presented a language-independent framework 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 approach can be seen as a systematic strategy for constructing proofs. The approach is proved sound and the tool implementing it is illustrated on an imperative-program example as well as on a more complex parallel program.

Related Work. There are several tools that perform program verification using symbolic execution. Some of them are more oriented towards finding bugs [9], while others are more oriented towards verification [11, 17, 20]. Several techniques are implemented to improve the performance of these tools, such as *bounded verification* [10] 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# [7] is a tool developed at Microsoft that extends C# with constructs for non-null types, preconditions, postconditions, and object invariants. Spec# comes with a sound programming methodology that permits specification and reasoning about object invariants even in the presence of 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 [15] and symbolic execution. The VeriFast tool [16] supports verification of single and multi-threaded C and Java programs annotated with preconditions and postconditions written in Separation Logic [21] 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 [22], which has beed used for verifying several challenging C programs such as the Schorr-Waite garbage collector. MatchC also uses the formalism of reachability logic for program specifications; it is, however, dedicated to a specific language.

By contrast, we focus on *language-independence*: given a a programming language defined in an algebraic/rewriting setting, we automatically generate the semantics for performing symbolic execution on that language, and build on this symbolic semantics for performing program verification. The soundness of our approach has also been proved. It relies on a Circularity Principle adapted to reachability logic, which has beed formulated in a different setting in [23].

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 proof system in Coq and proving its soundness with respect to the original proof system of RL (which have already been proved sound in Coq [22]). Secondly, `kcheck` must be enhanced to return, for any successfull execution, the rules of our system it has applied, and the substitutions it has used. From this information a Coq script is built that, if 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 approach to symbolic execution. Research report RR-8189, INRIA, Dec. 2012. Available at `http://hal.inria.fr/hal-00766220/PDF`.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

Lemma 1 (Unification by Matching) 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 $\sigma : var(t_1) \mapsto T_\Sigma(var(t_2))$ such that $t_1\sigma = t_2$ and such that $\sigma^{t_1}_{t_2} \triangleq \sigma \uplus id_{var(t_2)}$ is a most-general unifier of t_1, t_2.
Proof By induction on the structure of t_1. In the base case, $t_1 \in Var$ and $\sigma = (t_1 \mapsto t_2)$, thus, $\sigma^{t_1}_{t_2} = (t_1 \mapsto t_2) \uplus id|_{var(t_2)}$. Now, $\sigma^{t_1}_{t_2}$ is obviously a symbolic unifier of t_1, t_2. To show that $\sigma^{t_1}_{t_2}$ is most general, consider any concrete unifier of t_1, t_2, say, ρ. Then, (a) $t_1\sigma^{t_1}_{t_2}\rho = t_2\rho$ because $\sigma^{t_1}_{t_2}$ maps t_1 to t_2, and (b) $t_2\rho = t_1\rho$ because ρ is a concrete unifier. Thus, $t_1\sigma^{t_1}_{t_2}\rho = t_1\rho$. Moreover, for all $x \in var(t_2)$, $x\sigma^{t_1}_{t_2}\rho = x\rho$ since $\sigma^{t_1}_{t_2}$ is the identity on $var(t_2)$. Thus, for all $y \in var(t_1) \uplus var(t_2) (= \{t_1\} \uplus var(t_2))$, $y\sigma^{t_1}_{t_2}\rho = y\rho$, which proves the fact that $\sigma^{t_1}_{t_2}$ is a most general unifier (by taking $\eta = \rho$ in Definition 3 of unifiers).

For the inductive step, let $t_1 = f(t^1_1, \ldots, t^n_1)$ with $f \in \Sigma \setminus \Sigma^{Data}$, $n \geq 0$, and $t^1_1, \ldots, t^n_1 \in T_\Sigma(Var)$ for $i = 1, \ldots, n$. There are two subcases regarding t_2:

- t_2 is a variable. This is impossible, since t_2 should be of a data sort because it is a variable, and of a non-data sort because of the lemma's hypotheses.
- $t_2 = g(t^1_2, \ldots, t^m_2)$ with $g \in \Sigma$, $m \geq 0$, and $t^1_2, \ldots, t^m_2 \in T_\Sigma(Var)$. Let ρ be a concrete unifier of t_1, t_2, thus, $t_1\rho = f(t^1_1, \ldots, t^n_1)\rho = \mathcal{T}_f(t^1_1\rho, \ldots, t^n_1\rho) = f(t^1_1\rho, \ldots, t^n_1\rho) =_\mathcal{T} t_2\rho = \mathcal{T}_g(t^1_2\rho, \ldots, t^m_2\rho)$, 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 $t^i_1\rho = t^i_2\rho$ for $i = 1, \ldots, n$. The respective subterms t^i_1 and t^i_2 of t_1 and t_2 satisfy the hypotheses of our lemma, except maybe for the fact that t^i_1 may have have a data sort. There are again two cases:
  - if for some $i \in \{1, \ldots, n\}$, t^i_1 has a data sort then by the hypotheses of our lemma t^i_1 is a variable, and we let $\sigma^i \triangleq (t^i_1 \mapsto t^i_2)$ and $\sigma^{t^i_1}_{t^i_2} \triangleq \sigma_i \uplus id_{var(t^i_2)}$, which is a most-general unifier of t^i_1 and t^i_2, which is proved like in the base case;
  - otherwise, t^i_1 and t^i_2 satisfy all the the hypotheses of our lemma. We can then use the induction hypothesis and obtain substitutions $\sigma^i : var(t^i_1) \to T_\Sigma(var(t^i_2))$ such that $t^i_1\sigma^i = t^i_2$ for all $i = 1, \ldots, n$, and the corresponding most-general-unifiers $\sigma^{t^i_1}_{t^i_2}$ for t^i_1 and t^i_2, of the form $\sigma^{t^i_1}_{t^i_2} = \sigma^i \uplus id_{var(t^i_2)}$.

  Let $\sigma \triangleq \biguplus_{i=1}^n \sigma^i : var(t_1) \to T_\Sigma(var(t_2))$, which is a well-defined substitution thanks to the linearity of t_1. We obtain $t_1\sigma = t_2$ from $t^i_1\sigma^i = t^i_2$ for all $i = 1, \ldots, n$. Thus, $\sigma^{t_1}_{t_2} \triangleq \sigma \uplus id_{var(t_2)}$ is (obviously) a symbolic unifier of t_1, t_2; we only have to prove that it is most general. For this, we first note that the equality $\sigma^{t_1}_{t_2} = \biguplus_{i=1}^n \sigma^{t^i_1}_{t^i_2}$ also holds. Then, consider any concrete unifier ρ of t_1 and t_2, thus, $t^i_1\rho = t^i_2\rho$ for $i = 1, \ldots, n$. From the fact that all the $\sigma^{t^i_1}_{t^i_2}$ are most-general-unifiers of t^i_1 and t^i_2 for $i = 1, \ldots, n$, we obtain the existence of valuations η_i such that $\sigma^{t^i_1}_{t^i_2}\eta_i = \rho|_{(var(t^i_1) \uplus var(t^i_2))}$, for $i = 1, \ldots, n$. Then, $\eta \triangleq \biguplus_{i=1}^n \eta_i$, which coincides with ρ on $var(t_2)$ and is well-defined on $var(t_1)$ thanks to the linearity of t_1, and η has the property that $\sigma^{t_1}_{t_2}\eta = \rho$, which proves that $\sigma^{t_1}_{t_2}$ is a most general unifier of t_1 and t_2 and concludes the proof.

Lemma 2 If $\mathcal{S}$ is total and φ is derivable for $\mathcal{S}$ then $\mathcal{S}$ is a cover for φ. *Proof* Let $\varphi \triangleq \pi \wedge \phi$. Since φ is derivable for $\mathcal{S}$, the set of rules $\mathcal{S}_\pi \subseteq \mathcal{S}$ that match π is nonempty. Let

then $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}_\pi$. Since $\mathcal{S}$ is total, $\models \bigvee_{\pi_1 \wedge \phi' \Rightarrow \pi' \wedge \phi'' \in \mathcal{S}} (\phi' \wedge \phi'')$, and thus $\models \bigvee_{\pi_1 \wedge \phi' \Rightarrow \pi' \wedge \phi'' \in \mathcal{S}} (\phi' \wedge \phi'')\sigma_\pi^{\pi_1}$ holds.

Since the latter disjunction is a subformula of the larger disjunction in Def. 7: $\bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} (\phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, we obtain $\models \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} (\phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, hence, $\models \phi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} (\phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$ holds, which proves the lemma.

Before we prove the next lemma we need to recapp results about the original RL proof system, used in the sequel and proved in [22]:

- *Substitution*: $\mathcal{S} \vdash_G \varphi\theta \Rightarrow \varphi'\theta$, if $\theta : Var \rightarrow T_\Sigma(Var)$ 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 $\models \varphi \rightarrow \varphi'$ 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 3 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, $\Delta_{\mathcal{S}'}(\varphi) \triangleq \bigvee_{[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}'} [\varphi']_\sim} \varepsilon([\varphi']_\sim)$. Since $\mathcal{S}'$ is a cover for φ, by Remark 5, $\Delta_{\mathcal{S}'}(\varphi)$ is a nonempty disjunction. Using Definition 4 we obtain that each φ' is of the form $\varphi' = \pi_2\sigma_\pi^{\pi_1} \wedge \phi'$ for some $\alpha \triangleq (\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in S$ and ϕ' satisfying $\models \phi' \leftrightarrow (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1}$, where $\sigma_\pi^{\pi_1}$ is the most general symbolic unifier of π, π_1 built in the proof of Lemma 1. The projection of $\sigma_\pi^{\pi_1}$ on $var(\pi)$ is the identity, and the projection on $var(\pi_1)$ is a substitution of $var(\pi_1)$ matching π_1 on π. 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 as well, i.e., $\phi\sigma_\pi^{\pi_1} = \phi$.

Using the above characterisation for the patterns φ', we obtain that the choices of the ε operation in $\Delta_{\mathcal{S}'} \triangleq \bigvee_{[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}'} [\varphi']_\sim} \varepsilon([\varphi']_\sim)$ can be made such that

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

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:

$$\mathcal{S} \vdash_G \pi \wedge \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} (\phi \wedge \phi_1\sigma_\pi^{\pi_1} \wedge \phi_2\sigma_\pi^{\pi_1}) \Rightarrow \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} \pi_2\sigma_\pi^{\pi_1} \wedge (\phi \wedge \phi_1 \wedge \phi_2)\sigma_\pi^{\pi_1} \tag{5}$$

We know from (4) that the right-hand side of (5) 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 7, the validity of $\phi \rightarrow \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} (\phi_1 \sigma_\pi^{\pi_1} \wedge \phi_2 \sigma_\pi^{\pi_1})$, which proves ($\diamondsuit$) and the lemma.

Lemma 4 $\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff $\mathcal{S}^\Delta \models \varphi \Rightarrow \varphi'$. *Proof* ($\Rightarrow$)
From $\mathcal{S} \models \varphi \Rightarrow \varphi'$ we obtain that for each terminating configuration $\gamma \in \mathcal{T}$ and valuation ρ such that $(\gamma, \rho) \models \varphi$, there is $\gamma' \in \mathcal{T}$ and a path $\gamma \overset{\alpha^*}{\Longrightarrow}_{\mathcal{S}} \gamma'$, with $\alpha^* \in \mathcal{S}^*$, such that $(\gamma', \rho) \models \varphi'$. We prove by induction on the length of the sequence α^* that ($\spadesuit$): $\gamma \overset{\alpha^*}{\Longrightarrow}_{\mathcal{S}} \gamma'$, which implies the ($\Rightarrow$) direction.

The base case of ($\spadesuit$) is trivial. For the inductive step, we use the fact that each transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma_2$ is induced by semantical rule $\alpha \triangleq \pi \wedge \phi \Rightarrow \varphi \in \mathcal{S}$ with a valuation ρ. Then, the corresponding rule $\pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \in \mathcal{S}^\Delta$ induces a corresponding transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}^\Delta} \gamma_2$ with the same valuation ρ.

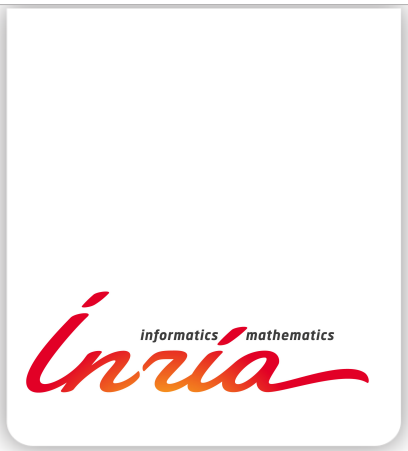
($\Leftarrow$) By analogy with the direct implication. The only difference is in the inductive step: each transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}^\Delta} \gamma_2$ is induced by semantical rule $\alpha^\Delta \triangleq \pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \in \mathcal{S}^\Delta$ with a valuation ρ. Then, the rule $\alpha \triangleq \pi \wedge \phi \Rightarrow \varphi \in \mathcal{S}$ induces a corresponding transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma_2$ with the same valuation ρ.

Lemma 5 A pattern φ is derivable for $\mathcal{S}$ iff φ is derivable for $\mathcal{S}^\Delta$. *Proof* By definition, $\varphi \triangleq \pi \wedge \phi$ is derivable for $\mathcal{S}$ means that $\Delta_{\mathcal{S}}(\varphi)$ is not the empty disjunction. This holds if and only if there exists a rule $\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}$ such that π_l matches π. The proof is finished by noting that π_l is also the left-hand side of the rule corresponding $\pi_l \Rightarrow \Delta_{\mathcal{S}}(\pi_l) \in \mathcal{S}^\Delta$.

Theorem 1 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 Lemmas 2 and 3. The second one, $(\mathcal{S} \cup G) \vdash \Delta_{\mathcal{S}}(\varphi_i) \Rightarrow \varphi_i'$ holds because all the rules of $\Vdash$ are derived rules of $\vdash$, thanks to Lemmas 2 and 3 again. 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 $\mathcal{S} \models G$.



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