Language-Independent Program Verification Using Symbolic Execution

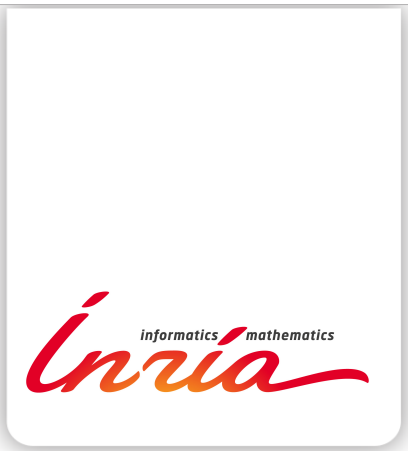
Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

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 — 22 pages

Abstract: In this paper we present an automatic and language-independent program verification approach based on symbolic execution. The specification formalism we consider is Reachability Logic, a language-independent logic that constitutes an alternative to Hoare logics. Reachability Logic has a sound and relatively complete deduction system, which offers a lot of freedom (but no guidelines) for constructing proofs. Hence, we propose symbolic execution as a strategy for proof construction. We show that, under reasonable conditions on the semantics of programming languages, our symbolic-execution based Reachability-Logic formula verification is sound. We present a prototype implementation of the resulting language-independent verifier as an extension of a generic symbolic execution engine that we are developing in the $\mathbb{K}$ framework. The verifier is illustrated on programs written in languages also formally defined in $\mathbb{K}$.

Key-words: Program Verification, Reachability Logic, Symbolic Execution.

* 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

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 d'appliquer les règles de déduction, mais qui n'offre 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 $\mathbb{K}$ *framework* et que nous illustrons sur des exemples de programmes écrits dans des langages formellement définis en $\mathbb{K}$.

Mots-clés : Vérification de programmes, *Reachability Logic*, exécution symbolique.

1 Introduction

Reachability Logic (RL) [27] is a language-independent logic for specifying program properties and for defining the formal operational semantics of programming languages. 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 program from a configuration where the program variables `a`, `b` are bound to non-negative values a, b, a configuration where the program variable `x` is bound to $\mathit{gcd}(a,b)$ is reached. Here, *gcd* is a mathematical description of the greatest-common-divisor (with $\mathit{gcd}(0,0)=0$ by convention) and *lookup* is a standard lookup function in associative maps.

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

Figure 1: Program `gcd`

Reachability Logic can also be used for defining the operational semantics of programming languages, such as that of the language in which the `gcd` program is written. The symbolic execution approach proposed in [5] automatically transforms a language's semantics such that programs can be executed with symbolic values. By running `gcd` with the symbolic values a, b for `a`,`b` one gets infinitely many computations, induced by the infinitely many iterations of the `while` loop. The first two ones are:

$$\langle\langle \text{gcd}\rangle_{\texttt{k}}\langle \texttt{a}\mapsto a\ \texttt{b}\mapsto b\rangle_{\texttt{env}}\rangle_{\texttt{cfg}} \wedge \mathit{true} \Rightarrow^* \langle\langle\cdot\rangle_{\texttt{k}}\langle \texttt{a}\mapsto a\ \texttt{b}\mapsto b\ \texttt{x}\mapsto a\ \texttt{y}\mapsto b\ \texttt{r}\mapsto 0\rangle_{\texttt{env}}\rangle_{\texttt{cfg}} \wedge \neg b > 0$$

$$\langle\langle \text{gcd}\rangle_{\texttt{k}}\langle \texttt{a}\mapsto a\ \texttt{b}\mapsto b\rangle_{\texttt{env}}\rangle_{\texttt{cfg}} \wedge \mathit{true} \Rightarrow^* \langle\langle\cdot\rangle_{\texttt{k}}\langle \texttt{a}\mapsto a\ \texttt{b}\mapsto b\ \texttt{x}\mapsto b\ \texttt{y}\mapsto a\,\%\,b\ \texttt{r}\mapsto a\,\%\,b\rangle_{\texttt{env}}\rangle_{\texttt{cfg}} \wedge b > 0 \wedge \neg a\,\%\,b > 0$$

Since the values of (`x`,`y`) in the final configurations are $(a,b),(b,a\%b),(a\%b,b\%(a\%b)),\ldots$ and knowing that $\mathit{gcd}(x,y) = \mathit{gcd}(y,x\,\%\,y)$, we could (unsoundly) conclude that the program satisfies the specification (1). This is because the semantics of the IMP language (presented in the paper), in which the `gcd` program is written, is not *total* as it does not have a rule for $\%\,0$.

Reachability Logic's proof system [27] is a set of seven inference rules that, for languages whose semantics satisfy certain reasonable well-formedness conditions, is sound and complete for proving reachability-logic formulas. The proof system is concise and elegant, but 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 guidelines for constructing proofs.

Contribution A language-independent framework and tool, based on symbolic execution, for proving properties of programs expressed in Reachability Logic. Our approach amounts to executing the proof system of RL in a certain systematic way, so that the symbolic execution tree for a given set of proof goals becomes a compact representation of the proof trees (in the RL deductive system) for the goals in question. The soundness of the proposed approach is based on

a so-called *circularity principle* for reachability-logic formulas. Soundness also 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 procedure in a prototype tool that extends our language-independent symbolic execution tool [5] which is part of the $\mathbb{K}$ framework [26]. The tool is illustrated on a parallel program written in a language also 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; and an example of a simple imperative language defined in $\mathbb{K}$. We also recap the main ingredients of generic framework for symbolic execution from [5], and present the proof system of Reachability Logic [27]. Section 3 contains the core contribution of the paper: an approach for verifying RL formulas by symbolic execution. We show that, under reasonable conditions, symbolic execution is a derived rule of the RL proof system and is thus sound by construction; and symbolic execution trees are compact representations of RL proof trees. We also give a circularity principle saying under which conditions it is sound to use RL goals 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. Section 4 describes a prototype verification tool based on our language-independent symbolic execution tool [5] 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 contains the proofs of the technical results in the paper. Our tool (with instructions of use) can be tried online on examples 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 The Ingredients of a Language Definition

In this section we identify the ingredients of language definitions in a algebraic and term-rewriting setting. A language $\mathcal{L}$ can be defined as a triple $(\Sigma, \mathcal{T}, \mathcal{S})$, consisting of a signature Σ, a model $\mathcal{T}$, and a set $\mathcal{S}$ of (semantical) rules.

Signature

Σ is a many-sorted algebraic signature, which includes at least a sort *Cfg* for *configurations* and a sort *Bool* for *constraint formulas*. We assume in this paper that the constraint formulas are Boolean terms built with a subsignature $\Sigma^{\mathsf{Bool}} \subseteq \Sigma$ including the boolean constants and operations[1] Σ may also include other subsignatures for other data sorts, depending on the language $\mathcal{L}$ (e.g., integers, identifiers, lists, maps,...). Let Σ^{Data} denote the subsignature of Σ consisting of all *data* sorts and their operations. We assume that the sort *Cfg* and the syntax of $\mathcal{L}$ are not data, i.e., they are defined in $\Sigma \setminus \Sigma^{\mathsf{Data}}$. 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.

[1] FOL formulas can be encoded as terms of sort *Bool* with quantifiers as constructors.

Model

We assume given a Σ^{Data}-model $\mathcal{D}$, which interprets the data sorts and their operations. Let $\mathcal{T}$ denote the free Σ-model generated by $\mathcal{D}$; thus, $\mathcal{D}$ can be seen as the "arithmetic-logic unit" over which the semantics of $\mathcal{L}$ is given. $\mathcal{T}$ interprets the non-data sorts as ground terms over the signature $(\Sigma \setminus \Sigma^{\mathsf{Data}}) \cup \mathcal{D}$.

Remark 1 *The exact mathematical construction of the model $\mathcal{T}$ is as follows. Let Alg_Σ denote the category of Σ-algebras and $Alg_{\Sigma^{\mathsf{Data}}}$ the category of the Σ^{Data}-algebras. The reduct functor $\_\restriction_{\Sigma^{\mathsf{Data}}} : Alg_\Sigma \to Alg_{\Sigma^{\mathsf{Data}}}$ sends a Σ-algebra M into Σ^{Data}-algebra $M\restriction_{\Sigma^{\mathsf{Data}}}$, where each data sort d and each operator in Σ^{Data} are interpreted as in M. The functor $\_\restriction_{\Sigma^{\mathsf{Data}}}$ has an left-adjoint functor $F : Alg_{\Sigma^{\mathsf{Data}}} \to Alg_\Sigma$. The model $\mathcal{T}$ is $F(\mathcal{D})$. We have the identity $\mathcal{T}\restriction_{\Sigma^{\mathsf{Data}}} = \mathcal{D}$.*

We also assume given a satisfaction relation $(\gamma, \rho) \models b$ between pairs consisting of concrete configurations $\gamma \in \mathcal{T}_{Cfg}$ and valuations $\rho : Var \to \mathcal{D}$, and constraint formulas $b \in T_{\Sigma,Bool}(Var)$. Since here we consider only constraints expressed as Boolean terms, $\models$ is defined by $(\gamma, \rho) \models b$ iff $\rho(b) = \mathcal{D}_{true}$. For simplicity, we often write in the sequel $true, false$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}$.

Rules

A set $\mathcal{S}$ of semantical rules. Each rule has the form $\varphi \Rightarrow \varphi'$ where φ, φ' are *patterns* over the set of variables Var. Patterns are formalised below.

Definition 1 (pattern over a given set of variables [27]) *An* elementary pattern *over a set of variables V is an expression of the form $\pi \wedge \phi$, where $\pi \in T_{\Sigma,Cfg}(V)$ is a* basic pattern *and $\phi \in T_{\Sigma,Bool}(V)$ is the pattern's* condition. *If $\gamma \in T_{Cfg}$ and $\rho : V \to \mathcal{T}$ we write $(\gamma, \rho) \models \pi \wedge \phi$ for $\gamma = \rho(\pi)$ and $\rho \models \phi$.*

The above definition is a particular case of a definition in [27]. There, a pattern is a first-order logic formula with configuration terms 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 $\pi \wedge true$. Sample patterns are $\langle\langle I_1 + I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \,/\, I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$.

Definition 2 (rule, transition system) *A* rule *is a pair of patterns over a set of variables Var, of the form $\varphi \Rightarrow \varphi'$. Any set $\mathcal{S}$ of rules defines a labelled transition system $(\mathcal{T}_{Cfg}, \Longrightarrow_{\mathcal{S}})$ such that $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff $\alpha \triangleq (\varphi \Rightarrow \varphi') \in \mathcal{S}$ and $\rho : Var \to \mathcal{T}$ are such that $(\gamma, \rho) \models \varphi$ 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 intensively used in research papers. 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), or blocks of statements. The attribute *strict* in some production rules means the arguments of the annotated expression/statement are evaluated before the expression/statement itself. If *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 a set of (possibly conditional) rewrite rules. The terms to which rules apply are called *configurations*. Configurations typically contain the program to be executed, together with any additional information required for program execution.

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

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

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

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

The structure of a configuration 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. 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 (cf. Figure 3). 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, e.g., $\texttt{a} \mapsto 3, \texttt{b} \mapsto 1$. The semantics of IMP is shown in Figure 4. Each rewrite rule from the semantics specifies how the configuration evolves when the first computation 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 shown in Figure 4 the semantics of IMP includes additional rules induced by the *strict* attribute. For the `if` statement, which is strict in its first argument, this argument is evaluated by executing the following rules:

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

Here, BE ranges over $\mathit{BExp} \setminus \{\mathit{false}, \mathit{true}\}$, B ranges over the Boolean values $\{\mathit{false}, \mathit{true}\}$, and $\square$ is a special variable, destined to receive the value of BE once it is computed, typically, by 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 ($\mathit{Int}, \mathit{Bool}, \mathit{AExp}, \ldots$) 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

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

Figure 4: $\mathbb{K}$ Semantics of IMP

the sort *Cfg*, the only constructor is $\langle\langle\_\rangle_{\mathsf{k}}\langle\_\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} : \mathit{Code} \times \mathit{Map}_{\mathit{Id},\mathit{Int}} \to \mathit{Cfg}$. The expression $\langle\langle X := I \curvearrowright C\rangle_{\mathsf{k}}\langle X \mapsto 0 \ \mathit{Env}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is a term of $T_{\mathit{Cfg}}(\mathit{Var})$, where X is a variable of sort *Id*, I is a 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 4) 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 multiset of maps $X \mapsto I$, where X ranges over identifiers *Id* and I over the integers. The value of an identifier X is 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, *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 in $\mathcal{D}$. For instance, the term $\texttt{if } 1 >_{\mathit{Int}} 0 \texttt{ then skip else skip}$ is intepreted as $\texttt{if } \mathcal{D}_{\mathit{true}} \texttt{ then skip else skip}$.

2.3 Symbolic Execution

We briefly recap our approach to symbolic execution from [5]: a new definition $(\Sigma^{\mathsf{s}}, \mathcal{T}^{\mathsf{s}}, \mathcal{S}^{\mathsf{s}})$ for a language $\mathcal{L}^{\mathsf{s}}$ is automatically generated from a given definition $(\Sigma, \mathcal{T}, \mathcal{S})$ of a language $\mathcal{L}$. The new language $\mathcal{L}^{\mathsf{s}}$ has the same syntax, and its semantics extends $\mathcal{L}$'s data domains with symbolic values and adapts the semantical rules of $\mathcal{L}$ to deal with the new domains. Then, the symbolic execution of $\mathcal{L}$ programs is the concrete execution of the corresponding $\mathcal{L}^{\mathsf{s}}$ programs, i.e., the application of the rewrite rules in the semantics of $\mathcal{L}^{\mathsf{s}}$. Building the definition of $\mathcal{L}^{\mathsf{s}}$ amounts to extending the signature Σ to a symbolic signature Σ^{s}, extending the Σ-algebra $\mathcal{T}$ to a Σ^{s}-algebra $\mathcal{T}^{\mathsf{s}}$, and turning the concrete rules $\mathcal{S}$ into symbolic rules $\mathcal{S}^{\mathsf{s}}$. We explain here the last

step; additional details are in [5].

We make the following *assumption*: the basic patterns in left-hand sides of rules do not contain operations on data, and the rules are left-linear. These assumptions can be made to hold by rule transformations as shown in [5]. Turning concrete rules into symbolic ones consists in transforming each rule $\varphi \Rightarrow \varphi' \in \mathcal{S}$, with $\varphi \triangleq \pi \wedge \phi$ and $\varphi' \triangleq \pi' \wedge \phi'$, into the following one:

$$\pi \wedge \psi \Rightarrow \pi' \wedge (\psi \wedge \phi \wedge \phi') \tag{2}$$

where $\psi \in \mathit{Var}$ is a fresh variable of sort *Bool* playing the role of a path condition. This means that symbolic rule are applied like concrete rules, except for the fact that the current path condition ψ is enriched with the rule's conditions ϕ, ϕ'.

Example

The first rule for `if` from the IMP semantics is transformed into the following rule in $\mathcal{S}^{\mathfrak{s}}$, by applying the various transformations from [5]:

$$\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge \psi \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge \psi \wedge (B = \mathit{true})$$

□

The triple ($\Sigma^{\mathfrak{s}}$, $\mathcal{T}^{\mathfrak{s}}$, $\mathcal{D}^{\mathfrak{s}}$) defines a language $\mathcal{L}^{\mathfrak{s}}$. Then, the transition system $(\mathcal{T}^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}, \Rightarrow_{\mathcal{S}^{\mathfrak{s}}})$ is defined using Definitions 1, 2 applied to $\mathcal{L}^{\mathfrak{s}}$. In [5] we prove that the symbolic transition system forward-simulates the concrete one, and that the concrete transition system backward-simulates the symbolic one. These two results then imply the naturally expected properties of symbolic execution. In this paper we use symbolic execution for the purpose of verifying RL formulas.

2.4 Reachability Logic

We recall the syntax and semantics of a subset of reachability logic (RL) [27]. The formulas we consider are the most commonly used ones for defining a language's semantics and for state properties about programs. Then its proof system and togethter with the soundness property [27] are briefly presented; these are essential in showing the correctness of our symbolic execution-based verification.

2.4.1 Syntax and Semantics

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.

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 $(\mathcal{T}_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}})$ 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 $(\mathcal{T}_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}})$.

t

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

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

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

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

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

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

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

Figure 5: Proof System for RL.

2.4.2 Proof System

We consider here the version of the reachability logic proof system described in [27]. 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 $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$. 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 [27]. $\varphi \wedge \phi$, for $\varphi \triangleq \bigvee_{i \in I} \pi_i \wedge \phi_i$, is a shortcut for $\bigvee_{i \in I} \pi_i \wedge (\phi_i \wedge \phi)$.

Definition 3 (weak well-definedness, [27]) *A set of rules $\mathcal{S}$ is* weakly well-defined *if for each $\varphi \Rightarrow \varphi' \in S$ and for all valuations $\rho : Var \rightarrow \mathcal{T}$ there exists a configuration γ such that $(\gamma, \rho) \models \varphi'$.*

Proposition 1 (soundness, [27]) *The deductive system of reachability logic 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 *relative completeness* result, but we are not using it here. Other results about the proof system, used in the sequel and proved in [27], are:

- *Substitution*: $\mathcal{S} \vdash_G \theta(\varphi) \Rightarrow \theta(\varphi')$, 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 $\mathcal{S}' \vdash \varphi \Rightarrow \varphi'$.

The proof system in Figure 5 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 the CaseAnalysis rule, which, thanks to the Consequence rule, can be arbitrary, provided their disjunction is logically equivalent to the premise.

3 Symbolic Execution for Reachability-Logic Verification

We show in this section how symbolic execution can be used for verifying RL formulas in a systematic manner. We show that, under the hypothesis of a *total* semantics, which amounts to saying that all situations are "covered" by the semantical rules, the tree generated by symbolic execution constitutes an (abbreviated) proof tree for a RL formula. The Circularity rule of symbolic execution, which allows one to prove programs with possibly unbounded loops by reusing goals to be proved as hypotheses, is here replaced by an instance of the Circularity Principle [25] for RL formulas. Using this principle amounts to performing symbolic execution in an extended semantics which includes the goals as new semantical rules. A goal can only be used in the symbolic execution of a given pattern only when it "covers" the pattern, which means that all symbolic successors of the pattern are computed by applying the goal as a rewrite rule. The following lemma characterises transitions in the symbolic transition system.

Lemma 1 (Symbolic Transitions) *For each transition $\varphi \overset{\alpha^{\mathsf{s}}}{\Longrightarrow} \varphi'$ in the symbolic transition system, with $\varphi \triangleq \pi \wedge \phi$ and $\alpha \triangleq \pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}$, there exists a substitution σ such that $\varphi = \sigma(\pi_l) \wedge \phi$ and $\varphi' = \sigma(\pi_r) \wedge (\phi \wedge \sigma(\phi_l) \wedge \sigma(\phi_r))$.*

3.1 Derivatives, Total Semantics, and Cover

The symbolic transition system $(\mathcal{T}^{\mathsf{s}}_{Cfg^{\mathsf{s}}}, \Rightarrow_{\mathcal{S}^{\mathsf{s}}})$ is useful for defining some notions that we use in the verification of reachability-logic formulas. A first, essential notion is that of *derivative* of an elementary pattern φ, which is is the disjunction of all elementary patterns that are targets of symbolic transitions starting in φ.

Definition 4 (Derivative) *For a set of semantical rules $\mathcal{S}$ and an elementary pattern φ, its* derivative *is defined by $\Delta_{\mathcal{S}}(\varphi) \triangleq \bigvee_{\alpha \in \mathcal{S}, \varphi \overset{\alpha^{\mathsf{s}}}{\Longrightarrow} \varphi'} \varphi'$.*

We note that if $\mathcal{S}$ is finite then $\Delta_{\mathcal{S}}(\varphi)$ is a finite disjunction because symbolic transition systems are *finitely branching*. Note that according to the Definition 4, the derivative of a pattern φ can be *false* (i.e., the empty disjunction).

Definition 5 (Derivable Pattern) *An elementary pattern $\varphi \triangleq \pi \wedge \phi$ is* derivable *for $\mathcal{S}$ if $\Delta_{\mathcal{S}}(\varphi)$ is a nonempty disjunction.*

The following notion of *total* semantics is essential for the soundness of RL-formula verification by symbolic execution. An example showing that the absence of totality leads to unsoundness was given in the introduction of the paper.

Definition 6 (Total Semantics) *We say that a set $\mathcal{S}$ of semantical rules is* total *if for each basic pattern π occurring in the left-hand side of a rule, the disjunction $\bigvee_{(\pi \wedge \phi \Rightarrow \varphi) \in \mathcal{S}} \phi$ is valid in FOL.*

Remark 2 *The semantics of* IMP *is not total because of the rules for division (and modulo):* $\langle\langle I_1 \mathbin{/} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0 \Rightarrow \langle\langle I_1 /_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$*, whose left-hand side pattern* $\langle\langle I_1 /_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$ *does not meet the condition of Definition 6. The semantics can be made total by adding a rule* $\langle\langle I_1 \mathbin{/} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 =_{Int} 0 \Rightarrow \langle error\rangle_{\mathsf{cfg}}$ *that leads divisions by zero into "error" configurations. We assume hereafter that the semantics of* IMP *has been transformed into a total semantics.*

Remark 3 *For basic patterns* π, π'*, let* $\sigma^{\pi}_{\pi'}$ *be a substitution such that* $\sigma^{\pi}_{\pi'}(\pi') = \pi$*. Since* $=$ *is syntactical equality, the substitution* $\sigma^{\pi}_{\pi'}$ *is unique if it exists.*

The notion of cover, defined below, is also essential for the soundness of RL-formula verification by symbolic execution, in particular, when a proof goal is circularly used as a hypothesis, which amounts to performing symbolic execution using the goal as a new semantical rule. 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* $\models \phi \rightarrow \bigvee_{\alpha \triangleq (\pi_\alpha \wedge \phi_\alpha \Rightarrow \varphi_\alpha) \in \mathcal{S}'} \sigma^{\pi}_{\pi_\alpha}(\phi_\alpha)$ *is called a* cover *of* φ*.*

Note that if $\mathcal{S}'$ is a cover for $\varphi \triangleq \pi \wedge \phi$, the disjunction $\bigvee_{\alpha \triangleq (\pi_\alpha \wedge \phi_\alpha \Rightarrow \varphi_\alpha) \in \mathcal{S}'} \sigma^{\pi}_{\pi_\alpha}(\phi_\alpha)$ is nonempty, otherwise the validity $\models$ in Definition 7 would not hold (since an empty disjunction is *false*). In particular, there is at least one rule in $\mathcal{S}'$ that matches π, and, as a consequence, the derivative $\Delta'_{\mathcal{S}}(\varphi)$ is a nonempty disjunction. The next result exhibits a simple and useful example of the notion of cover.

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 new derived rule of the RL proof system:

Theorem 1 *If* $\mathcal{S}' \subseteq \mathcal{S}$ *is a cover for* φ*, then* $\mathcal{S} \vdash_G \varphi \Rightarrow \Delta_{\mathcal{S}'}(\varphi)$ *for any set* G*.*

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

3.2 Weakly Well-Definable Semantics

Before we can use the soundness of the RL deductive system we need to deal with the issue that operational semantics are not always weakly well-defined (Def. 3) as required by the RL soundness result. 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 $\langle\langle I_1 \mathbin{\%} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 =_{Int} 0 \Rightarrow \langle error\rangle_{\mathsf{cfg}}$ in order to make the semantics total (cf. Remark 2), the semantical of division can now equivalently rewritten using just one (reachability-logic) rule:

$$\langle\langle I_1 \mathbin{\%} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0 \Rightarrow (\langle\langle I_1 \%_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \wedge I_2 =_{Int} 0) \vee (\langle error\rangle_{\mathsf{cfg}}) \tag{3}$$

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 and ensures soundness of the RL deduction system. We note that in this case $\mathcal{S}^\Delta$ is also (trivially) total, which is required by the soundness of our approach as shown in the example in the introduction. A few other useful properties of the $\mathcal{S}^\Delta$ construction are given below.

Lemma 3 *$\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff $\mathcal{S}^\Delta \models \varphi \Rightarrow \varphi'$.*

Lemma 4 *A pattern φ is derivable for $\mathcal{S}$ if and only if φ is derivable for $\mathcal{S}^\Delta$.*

3.3 Reachability-Logic Verification 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, say, $G = \{\varphi_i \Rightarrow \varphi'_i \mid i = 1, \ldots, n\}$. We now have almost all ingredients for proving $\mathcal{S} \models G$ by symbolic execution.

Definition 10 *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 $\Delta_\mathcal{S}(G)$ denotes the set $\{\Delta_\mathcal{S}(\varphi) \Rightarrow \varphi' \mid \varphi \Rightarrow \varphi' \in G\}$. If Φ is a set of RL formulas, then $\mathcal{S} \vdash \Phi$ denotes the conjunction $\bigwedge_{\varphi \Rightarrow \varphi' \in \Phi} \mathcal{S} \vdash \varphi \Rightarrow \varphi'$, and $\mathcal{S} \models \Phi$ denotes $\bigwedge_{\varphi \Rightarrow \varphi' \in \Phi} \mathcal{S} \models \varphi \Rightarrow \varphi'$.*

The last result we shall be using in order to prove $S \models G$ by symbolic execution says the formulas G themselves may be used as (circular) hypotheses:

Theorem 2 (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 2 with $\mathcal{S}^\Delta$ instead of $\mathcal{S}$, and, under the same hypotheses regarding the derivability of G, one can deduce $\mathcal{S} \models G$, thanks to Lemmas 3 and 4.

Theorems 1 and 2 together with the soundness of the derived deduction rules of RL mentioned in Section 2.4 constitute the formal basis for the soundness of our implementation of verification of RL formulas based on symbolic execution.

This verification amounts to searching proofs for $\mathcal{S} \cup G \vdash \Delta_\mathcal{S}(G)$. An important observation is that we do not need to use the Circularity rule of the RL proof system (Figure 5) because we give all circularities G right from the beginning in the Circularity Principle. So, the difficult and tricky task of detecting circularities during the proof process is avoided. It could be claimed that finding an adequate set of the circularities G, allowing the Circular Principle to be used for proving $\mathcal{S} \models G$, is as difficult as the finding them during the proving process. This is only true in theory: in practice, the main advantage of our approach is that it is automatic and it can be tried on several guesses for the set G. Moreover, when a proof fails, one may check the path conditions returned by symbolic execution to guess what new goals should be added for progressing in the proof.

We search for proofs of $\mathcal{S} \cup G \vdash \Delta_\mathcal{S}(G)$ in the deductive system from Figure 6, which is a subsystem of the RL proof system. This is because all its rules are either deductive rules of RL (like Transitivity and CaseAnalysis) or derived rules, like Implication. SymbolicStep is a also

t

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

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

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

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

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

Figure 6: Proof System for $\mathcal{S} \cup G \vdash \Delta_{\mathcal{S}}(G)$.

derived rule, which is shown using $\mathcal{S} \cup G \vdash \varphi \Rightarrow \Delta_{\mathcal{S}}(\varphi)$ (cf. Corollary 1) and the *Monotony* derived rule of RL. Finally, CircularHypothesis is also a derived rule thanks to Theorem 1.

Example We show how the RL formula (1) is proved using the deductive system in Figure 6, which amounts to verifying the `gcd` program. For this, we consider the two following additional formulas, where `while` denotes the program fragment consisting of the `while` loop, and `body` denotes the body of the loop:

$$\langle\langle \texttt{while} \rangle_{\texttt{k}} \langle \texttt{a} \mapsto a\ \texttt{b} \mapsto b\ \texttt{x} \mapsto x\ \texttt{y} \mapsto y\ \texttt{r} \mapsto r \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} \wedge gcd(a,b) = gcd(x,y) \wedge x \geq 0 \wedge y \geq 0 \Rightarrow$$
$$\langle\langle \cdot \rangle_{\texttt{k}} \langle \texttt{a} \mapsto a\ \texttt{b} \mapsto b\ \texttt{x} \mapsto x'\ \texttt{y} \mapsto y'\ \texttt{r} \mapsto r' \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} \wedge gcd(a,b) = gcd(x',y') \wedge x' \geq 0 \wedge y' = 0 \quad (4)$$
$$\langle\langle \texttt{body}\ \cdots \rangle_{\texttt{k}} \langle \texttt{a} \mapsto a\ \texttt{b} \mapsto b\ \texttt{x} \mapsto x\ \texttt{y} \mapsto y\ \texttt{r} \mapsto r \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} \wedge gcd(a,b) = gcd(x,y) \wedge x > 0 \wedge y \geq 0 \Rightarrow$$
$$\langle\langle \cdot \rangle_{\texttt{k}} \langle \texttt{a} \mapsto a\ \texttt{b} \mapsto b\ \texttt{x} \mapsto x'\ \texttt{y} \mapsto y'\ \texttt{r} \mapsto r' \rangle_{\texttt{env}} \rangle_{\texttt{cfg}} \wedge gcd(a,b) = gcd(x',y') \wedge x' \geq 0 \wedge y' \geq 0 \quad (5)$$

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), (4), and (5), and apply the deductive system in Figure 6 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 (4). Next, the Implication rule is used to prove that the pattern resulting from applying the formula (4) 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 (4) 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 (4), and then Transitivity builds a proof of (4) 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 (5), then Implication is used to prove that the current pattern implies the right-hand side of the formula (4), and finally Transitivity builds a proof of (4) from these rule applications.

– the formula (5) 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), (4), and (5), and, in particular, of the fact that the `gcd` program meets its specification (1). Note how the proofs of all goals used symbolic execution as well other goals as circular hypotheses. □

4 A Prototype Implementation

In this section we describe the implementation of a tool `kcheck`, which uses the deductive system that we have defined in Figure 6 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 framework [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 $\mathbb{K}$: Scheme [22], Verilog [20], C [13], Python [14], Java [7], OCaml [18]. An example of a $\mathbb{K}$ definition can be found in Section 2.2. The framework also includes support for symbolic execution. One can execute programs having symbolic inputs using a transformed definition of a language which is generated automatically from its original $\mathbb{K}$ definition [5]. The result of the symbolic execution consists in a set of logical constraints corresponding to different program paths. In terms of implementation, our prototype reuses some components of the $\mathbb{K}$ framework: parsing, compilation steps, support for symbolic execution, and the connections to the Maude's [21] model-checker and the Z3 SMT solver [12]. Since our approach is parametric in the formal definition of the language, `kcheck` can also benefit in the future from the new language definitions.

The tool completes two stages during its execution: it 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 $\mathcal{S}^{\mathfrak{s}} \cup G$, where $\mathcal{S}^{\mathfrak{s}}$ is defined as in Section 2.3. If $\mathcal{S}^{\mathfrak{s}}$ is weakly well-definable (Def. 9) but not weakly well-defined (Def. 3) then we use $(\mathcal{S}^{\mathfrak{s}})^{\Delta}$ instead of $\mathcal{S}^{\mathfrak{s}}$. 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 proof system in Fig. 6.

Computing all the successors of a pattern by applying rules from $\mathcal{S}^{\mathfrak{s}}$ 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 $\mathcal{S}^{\mathfrak{s}}$. 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 is used to build proof trees in the deductive system in Fig. 6 from symbolic execution trees.

We have used `kcheck` to prove `gcd.imp` (Figure 1) as sketched in Example 3.3. The tool has also been used to prove all the IMP programs from [4]. In the rest of the section we illustrate the

```
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 7: `FIND` program.

verification of a parallel program in order to show that our approach is language-independent and can deal with nontrivial proofs.

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 7 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 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:

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

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 $\langle\langle \mathit{S1} \rangle_{\mathsf{k}} \langle\rho\rangle_{\mathsf{env}} \langle \mathit{id}_1 \rangle_{\mathsf{id}} \rangle_{\mathsf{th}}$ and $\langle\langle \mathit{S2} \rangle_{\mathsf{k}} \langle\rho\rangle_{\mathsf{env}} \langle \mathit{id}_2 \rangle_{\mathsf{id}} \rangle_{\mathsf{th}}$, 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 8 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 reachability formulas that were constructed from the above pre/post conditions and invariants. Finally we show the proofs given by `kcheck`.

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)	$\langle\langle \mathtt{S1}\|\|\mathtt{S2};\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St\rangle_{\mathsf{st}} \wedge p_1 \wedge p_2$	$\Rightarrow$	$\langle\langle\cdot\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{th}}\langle St'\rangle_{\mathsf{st}} \wedge q_1 \wedge q_2$
(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 8: 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 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 reachability rules (3), (5), and (7), where CaseAnalysis splits the proof in two branches. The proof tree for the reachability formula (8), corresponding to the specification of `FIND`, has a single branch because it uses circularities, which do not split the proof tree, making the proof tree more compact.

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 reachability 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.

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 also forgotten in [4].

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 can be used for initial debugging of programs before they are verified.

5 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, 17, 23]. 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 [11]. 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 important 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: allocated memory is eventually freed, opened files are eventually closed, file reading and writing is allowed only on opened files, etc.

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 [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 [24] 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 formalism of reachability logic for program specifications; it is, however, dedicated to a specific language.

Our approach addresses the verification problem from a different angle. We focus on *language-*

independence: given a formal definition of a programming language whose semantics is defined by rewriting, 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 been proved. It relies on a Circularity Principle adapted to reachability logic, which has beed formulated in a different setting in [25].

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.

6 Conclusion 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.

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, and then use the tool on the "real" language definitions in the $\mathbb{K}$ framework [7, 18].

Another future research direction is embedding our verifier in a proof assistant such as Coq [1] in order to allow a degree of user guidance in proofs. We envisage to encode the proof system in Figure 6 in Coq and to use our tool `kcheck` as an external proof constructor. Coq requires from external tools a proof term, which in re-checks in order to accept the proof; thus, `kcheck` must be enhanced to build proof terms in the Coq formalism from the proof trees it generates.

We are also planning to combine program verification with the program equivalence framework we developed in [19]. The idea is to perform verification on an abstract program, and, via property-preserving equivalences, to ensure that the properties also hold on more concrete versions of the verified program.

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 at `http://hal.inria.fr/hal-00766220/`.

[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] 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.

[12] 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.

[13] 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.

[14] D. Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

[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] D. Lazăr. OCaml semantics in $\mathbb{K}$. `https://github.com/kframework/ocaml-semantics`.

[19] D. Lucanu and V. Rusu. Program equivalence by circular reasoning. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *LNCS*, pages 362–377. Springer, 2013.

[20] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010.

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

Appendix A: Proofs

Lemma 1. For each transition $\varphi \stackrel{\alpha^{\mathfrak{s}}}{\Longrightarrow} \varphi'$ in the symbolic transition system, with $\varphi \triangleq \pi \wedge \phi$ and $\alpha \triangleq \pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}$, there exists a substitution σ such that $\varphi = \sigma(\pi_l) \wedge \phi$ and $\varphi' = \sigma(\pi_r) \wedge (\phi \wedge \sigma(\phi_l) \wedge \sigma(\phi_r))$.
Proof Since $\alpha \triangleq \pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r$, the corresponding symbolic rule $\alpha^{\mathfrak{s}}$ is of the form (2), i.e, $\alpha^{\mathfrak{s}} = (\pi_l \wedge \psi \Rightarrow \pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r))$ where ψ is a fresh Boolean variable. The transition $\varphi \stackrel{\alpha^{\mathfrak{s}}}{\Longrightarrow} \varphi'$ is obtained by rewriting $\varphi \triangleq \pi \wedge \phi$ with $\alpha^{\mathfrak{s}}$, hence, there is a substitution $\sigma \cup (\psi \to \phi)$ such that $(\sigma \cup (\psi \to \phi))(\pi_l \wedge \psi) = \pi \wedge \phi$ and $\varphi' = (\sigma \cup (\psi \to \phi))(\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r))$. The conclusion of the lemma is obtained by applying the substitution $\sigma \cup (\psi \to \phi)$ in the last two identities. □

Lemma 2. If $\mathcal{S}$ is total and φ is derivable for $\mathcal{S}$ then $\mathcal{S}$ is a cover for φ.
Proof Since $\varphi \triangleq \pi \wedge \phi$ is derivable for $\mathcal{S}$, the derivative $\Delta_{\mathcal{S}}(\varphi)$ is a nonempty disjunction, hence, the set of rules $\mathcal{S}_\pi \subseteq \mathcal{S}$ that match π is nonempty. Consider then any rule $\alpha_0 \triangleq (\pi_{\alpha_0} \wedge \phi_{\alpha_0} \Rightarrow \varphi_{\alpha_0}) \in \mathcal{S}_\pi$. Since $\mathcal{S}$ is total, the disjunction $\bigvee_{(\pi_{\alpha_0} \wedge \phi' \Rightarrow \varphi') \in \mathcal{S}} \phi'$ is valid, hence, so is $\bigvee_{(\pi_{\alpha_0} \wedge \phi' \Rightarrow \varphi') \in \mathcal{S}} \sigma^{\pi}_{\pi_{\alpha_0}}(\phi')$. Since the latter disjunction is a subformula of the larger disjunction $\bigvee_{\alpha \triangleq (\pi_\alpha \wedge \phi_\alpha \Rightarrow \varphi_\alpha) \in \mathcal{S}} \sigma^{\pi}_{\pi_\alpha}(\phi_\alpha)$, we have $\models \bigvee_{\alpha \triangleq (\pi_\alpha \wedge \phi_\alpha \Rightarrow \varphi_\alpha) \in \mathcal{S}} \sigma^{\pi}_{\pi_\alpha}(\phi_\alpha)$, hence, $\models \phi \to \bigvee_{\alpha \triangleq (\pi_\alpha \wedge \phi_\alpha \Rightarrow \varphi_\alpha) \in \mathcal{S}} \sigma^{\pi}_{\pi_\alpha}(\phi_\alpha)$.

Hence, using Definition 7, $\mathcal{S}$ is a cover for φ, which proves the lemma.

Theorem 1. If $\mathcal{S}' \subseteq \mathcal{S}$ is a cover for φ, then $\mathcal{S} \vdash_G \varphi \Rightarrow \Delta_{\mathcal{S}'}(\varphi)$ for any set G.
Proof Let $\varphi \triangleq \pi \wedge \phi$. By Definition 4, $\Delta_{\mathcal{S}'}(\varphi) \triangleq \bigvee_{\alpha \in \mathcal{S}', \varphi \stackrel{\alpha^{\mathfrak{s}}}{\Longrightarrow} \varphi'} \varphi'$. Since $\mathcal{S}'$ is a cover for φ (Definition 7), $\Delta_{\mathcal{S}'}(\varphi)$ is a nonempty disjunction. Using Lemma 1 we obtain that each φ' is of the form $\varphi' = \sigma(\pi_r) \wedge (\phi \wedge \sigma(\phi_l) \wedge \sigma(\phi_r))$, for some rule $\alpha \triangleq (\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r) \in \mathcal{S}'$ and substitution σ such that $\sigma(\pi_l) = \pi$. We use the notation $\sigma^{\pi_l}_{\pi}$ for this substitution and note that it is unique (cf. Remark 3).

Using the above characterisation for the end-patterns φ' of symbolic transitions and the definition of derivatives:

$$\Delta_{\mathcal{S}'}(\varphi) \equiv \bigvee_{(\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r) \in \mathcal{S}'} \sigma^{\pi_l}_{\pi}(\pi_r) \wedge (\phi \wedge \sigma^{\pi_l}_{\pi}(\phi_l) \wedge \sigma^{\pi_l}_{\pi}(\phi_r)) \tag{6}$$

On the other hand, by using the derived rules of the RL proof system: *Substitution* with the rule $\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r$ and substitution $\sigma^{\pi_l}_{\pi}$, and *LogicalFraming* with the FOL formula ϕ, we get $\mathcal{S} \vdash_G \sigma^{\pi_l}_{\pi}(\pi_l \wedge \phi_l) \wedge \phi \Rightarrow \sigma^{\pi_l}_{\pi}(\pi_r \wedge \phi_r) \wedge \phi$, and, using the fact that $\sigma^{\pi_l}_{\pi}(\pi_l) = \pi$ together with the Consequence rule of RL, and remembering that FOL patternless formulas distribute over patterns, we obtain

$$\mathcal{S} \vdash_G \pi \wedge (\phi \wedge \sigma^{\pi_l}_{\pi}(\phi_l)) \Rightarrow \sigma^{\pi_l}_{\pi}(\pi_r) \wedge (\phi \wedge \sigma^{\pi_l}_{\pi}(\phi_r))$$

for every semantical rule $\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}'$. Next, using *LogicalFraming* with the above derivation and with $\sigma^{\pi_l}_{\pi}(\phi_l)$ and then Consequence, we obtain

$$\mathcal{S} \vdash_G \pi \wedge (\phi \wedge \sigma^{\pi_l}_{\pi}(\phi_l)) \Rightarrow \sigma^{\pi_l}_{\pi}(\pi_r) \wedge (\phi \wedge \sigma^{\pi_l}_{\pi}(\phi_l) \wedge \sigma^{\pi_l}_{\pi}(\phi_r))$$

for every rule $\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}'$. Using CaseAnalysis and Consequence:

$$\mathcal{S} \vdash_G \pi \wedge (\phi \wedge \bigvee_{\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}'} \sigma^{\pi_l}_{\pi}(\phi_l)) \quad \Rightarrow \bigvee_{\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}'} \sigma^{\pi_l}_{\pi}(\pi_r) \wedge (\phi \wedge \sigma^{\pi_l}_{\pi}(\phi_l) \wedge \sigma^{\pi_l}_{\pi}(\phi_r))$$

We know from (6) that the right-hand side of the above 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: $(\phi \wedge \bigvee_{\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}'} \sigma_{\pi}^{\pi_l}(\phi_l))$ is logically equivalent to ϕ in FOL.

Finally, since $\mathcal{S}'$ is a cover for φ, we obtain, using Definition 7, the validity of $\phi \rightarrow \bigvee_{\pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r \in \mathcal{S}'} \sigma_{\pi}^{\pi_l}(\phi_l)$, which proves ($\diamondsuit$) and the lemma.

Lemma 3. $\mathcal{S} \models \varphi \Rightarrow \varphi'$ 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 direct ($\Rightarrow$) implication.

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.

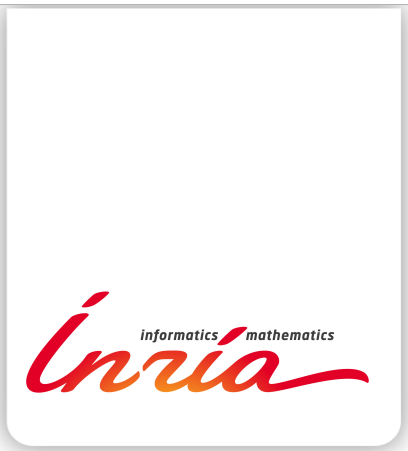
($\Leftarrow$) By analogy with the direct implication. The only difference is in the inductive step: each transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}^{\Delta}} \gamma_2$ is induced by semantical rule $\alpha^{\Delta} \triangleq \pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \in \mathcal{S}^{\Delta}$ 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 4. A pattern φ is derivable for $\mathcal{S}$ if and only if 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 2. 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 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 Theorem 1 and Lemma 2. The second one, $(\mathcal{S} \cup G) \vdash \Delta_{\mathcal{S}}(\varphi_i) \Rightarrow \varphi_i'$ is a hypothesis of the theorem. 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 $\mathcal{S}$ 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