PROGRAM LOGICS AND THEIR APPLICATIONS

ANDREI ARUSOAIE, ŞTEFAN CIOBÂCĂ, DOREL LUCANU, GRIGORE ROŞU, VLAD RUSU and TRAIAN-FLORIN ŞERBĂNUŢĂ

Communicated by Marius Iosifescu

The semantics of programming languages is being studied for a long time, but it is still lacking a unified framework where this can be easily formally specified and then the formal definition is used for experimenting (*e.g.* in the design process), language analysis (to check language properties), program verification, analysis, and automated testing, and for (correct) implementation. This paper surveys a part of the results obtained by the joint effort made for finding such a framework by the research groups Formal System Laboratory (FSL) from the University of Illinois in Urbana-Champaign (USA), Formal Methods in Software Engineering (FMSE) from the Alexandru Ioan Cuza University of Iaşi (Romania), and Dreampal from INRIA Lille Nord Europe (France).

AMS 2010 Subject Classification: 68N30.

Key words: matching logic, reachability logic, program logics, formal reasoning, program verification, symbolic execution, program equivalence.

1. INTRODUCTION

In this paper, we present a unified framework of logics suitable for formally defining programming languages and for reasoning about programs. We discuss *Reachability Logic* (RL) [14, 20], which is employed in formally specifying programming languages semantics and program properties. RL formulae are pairs of *Matching Logic* (ML) formulae [13, 16]. ML is a logic for reasoning statically about the structure of program configurations and RL captures the dynamic evolution of such configuration over time. Both logics, ML and RL, are endowed with sound and complete deductive systems. We use such deductive systems in the program analyses that we develop in a *language-independent setting*: symbolic execution, program verification, and program equivalence verification. The various deductive systems often have a *circular* (or *coinductive*) nature: a set of formulas under proof can use each other during the proof process; a formula may even use itself in its own proof, of course, in a restricted manner in order to avoid vicious-circle reasoning.

Our running example language is the WH language (because its basic construct is WHile loop), which includes simple arithmetic expressions and the assignment, while, and sequential composition statements. The syntax of WH is given in Fig. 1. We illustrate the use of RL in defining the operational semantics of WH and in specifying WH programs. We then illustrate symbolic execution and verification of WH programs based on the RL semantics of the language.

```
AExp ::=
       Int
     | Id
     | AExp "+" AExp
     > AExp "/" AExp
     | (AExp)            [bracket]

BExp ::=
       Bool
     | AExp "<" AExp
     | AExp "==" AExp
     | (BExp)            [bracket]

Stmt ::=
       Id "=" AExp ";"
     | "while" "(" BExp ")" Stmt
     | "{" Stmt "}"
     | Stmt Stmt [right]
```

Fig. 1. The syntax of WH.

The structure of the paper is as follows. In Section 2 we introduce Matching Logic and in Section 3, we introduce Reachability Logic formulas as pairs of Matching Logic formulas. Together, the two logics are used in a generic formal notion of programming language definition. The next section is dedicated to the all-paths version of Reachability Logic, a logic suitable for non-deterministic programs. Sound and relative complete deductive systems are given for this logic. Section 5 presents the symbolic execution of programs and shows how a two-rule deductive system can formalise it in a language-independent manner. A third rule makes the proof system circular and enables the verification of Reachability-Logic formulas based on symbolic execution as defined. Next, Section 6 shows an approach that we proposed for proving a certain notion of program equivalence: programs in different languages can be proved equivalent, using a dedicated proof system that also has a circular nature, and that operates on programs from a programming that "aggregates" the two languages of interest. In Section 7 we present the $\mathbb{K}$ framework, in which most of our language semantics and program analysis approaches are implemented. Concluding remarks are given in Section 8.

2. MATCHING LOGIC

The underlying logic of our framework is Matching Logic (ML) [13]. ML is a static logic of configurations, the main ingredient for the operational semantics of the program languages. ML uniformly generalises several logics that were intensively used for programming language semantics and/or program analysis, *e.g.*, propositional logic, algebraic specification, and separation logic. For the sake of presentation, we include in this paper only a simplified version of ML, namely the topmost fragment of ML.

Since topmost ML is a methodological (many-sorted) variant of the first-order logic (FOL), we briefly recall the syntax and the semantics of FOL.

Definition 1 (Many-Sorted First-Order Logic (FOL)). Given a set S of sorts, a *first-order signature* (Σ, Π) consists of a $S^* \times S$-sorted set Σ of function symbols (i.e., a many-sorted signature), and a S^*-sorted set Π of predicate symbols. A (Σ, Π)-*model* consists of a Σ-algebra $\mathcal{T}$ and a subset $\mathcal{T}_p \subseteq \mathcal{T}_{s_1} \times \cdots \times \mathcal{T}_{s_n}$ for each $p \in \Pi_{s_1 \ldots s_n}$. Let *Var* denote a S-sorted set of variables. The set of (Σ, Π)-*formulas* is defined by

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

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

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

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

The other first-order formulas (including disjunction, implication, equivalence, universal quantifier, ...) are defined as syntactic sugar over existing formulae in the usual way.

Example 1. The BNF grammar given by Fig. 1 defines a FOL signature $(\Sigma(\mathrm{WH}), \Pi(\mathrm{WH}))$, where: $S = \{\mathtt{Id}, \mathtt{Int}, \mathtt{Bool}, \mathtt{AExp}, \mathtt{BExp}, \mathtt{Stmt}\}$ is the set of non-terminals; each grammar production of sort `AExp` or `Stmt` defines an operation in $\Sigma(\mathrm{WH})$, *e.g.*, `AExp ::= AExp "+" AExp` defines the operation $\_+\_ :$ `AExp AExp` $\to$ `AExp`; each operation of sort `BExp` defines a predicate in $\Pi(\mathrm{WH})$, *e.g.*, `BExp ::= AExp "<" AExp` defines the predicate $\_<\_ :$ `AExp AExp`.

The $(\Sigma(\mathrm{WH}), \Pi(\mathrm{WH}))$-model we consider in this paper interprets `Id` as the set of identifiers (alphanumeric strings starting with a letter), `Int` as the set of integers $\mathbb{Z}$, `Bool` as the set of boolean values $\mathbb{B}$, and the function and predicate symbols in the usual way.

Definition 2 (ML signature). An ML signature is a triple $\Phi = (\Sigma, \Pi, \mathit{Cfg})$, where (Σ, Π) is a first-order signature and *Cfg* is a distinguished sort in Σ for *configurations*. The set of *ML formulas* over Φ and the variables *Var* is defined by

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

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

Example 2. A possible configuration for WH is represented by an operation $\langle\_\rangle\langle\_\rangle : \mathit{Stmt}\ \mathit{State} \to \mathit{Cfg}$, where *State* is the sort of finite sets of pairs *variable-name* $\mapsto$ *value*. An example of ML-formula is

$$\langle \mathtt{while}(\mathtt{x} < 7)\{\mathtt{y} = \mathtt{y} + \mathtt{x}; \mathtt{x} = \mathtt{x} + 1;\}\rangle\langle \mathtt{x} \mapsto a, \mathtt{y} \mapsto b\rangle \wedge b <_{\mathrm{Int}} a,$$

where a and b are variables of sort `Int`. Its intended meaning is that the `while` statement is executed in a state where the program variable `x` has the value a, `y` has the value b, and the two values satisfy the constraint $b <_{\mathrm{Int}} a$. By $\_ <_{\mathrm{Int}} \_$ we denote the predicate "less than" over integers, which is a part of the FOL model we consider. This convention is used for all predicates and operations over integers.

Definition 3 (ML Model). A model for ML signature $\Phi = (\Sigma, \Pi, \mathit{Cfg})$ is a (Σ, Π) first-order model $\mathcal{T}$. *Concrete configurations* (or simply *configurations*) are elements of $\mathcal{T}_{\mathit{Cfg}}$, i.e., $\mathcal{T}$-interpretations of ground terms of sort *Cfg*.

Example 3. An example of concrete configuration in $\mathcal{T}$(WH), the model we consider for WH, is

$$\langle \mathtt{while}(\mathtt{x} < 7)\{\mathtt{y} = \mathtt{y} + \mathtt{x}; \mathtt{x} = \mathtt{x} + 1;\}\rangle\langle x \mapsto 2, y \mapsto 0\rangle$$

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

Definition 4 (ML Satisfaction). The satisfaction relation $\models$ relates pairs (γ, ρ), where $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ and $\rho : \mathit{Var} \to \mathcal{T}$, with Φ-formulas φ. For basic patterns π, $(\gamma, \rho) \models \pi$ iff $\gamma = \pi\rho$. For the remaining ML constructions satisfaction is defined as expected, *e.g.*, $(\gamma, \rho) \models \exists X\varphi$ iff $(\gamma, \rho') \models \varphi$ for some $\rho' : \mathit{Var} \to \mathcal{T}$ such that $x\rho = x\rho'$ for all $x \in \mathit{Var} \setminus X$. If φ is an ML formula then $[\![\varphi]\!]$ denotes the set of concrete configurations $\{\gamma \mid (\gamma, \rho) \models \varphi \text{ for some } \rho\}$.

Example 4. If γ is the configuration in Example 3 and ρ is s.t. $a\rho = 2$ and $b\rho = 0$, then $(\gamma, \rho) \models \varphi$, where φ is the ML formula in Example 2. On the other hand, if ρ' is such that $a\rho = 2$ and $b\rho = 7$ then $(\gamma, \rho) \not\models \varphi$ because $7 \not<_{\mathrm{Int}} 2$. If $\gamma' = \langle \mathtt{y} = \mathtt{y} + \mathtt{x}; \mathtt{x} = \mathtt{x} + 1;\rangle\langle x \mapsto 2, y \mapsto 0\rangle$, then $(\gamma', \rho) \not\models \varphi$ because $\langle \mathtt{while}(\mathtt{x} < 7)\{\mathtt{y} = \mathtt{y} + \mathtt{x}; \mathtt{x} = \mathtt{x} + 1;\}\rangle\langle \mathtt{x} \mapsto a, \mathtt{y} \mapsto b\rangle\rho \neq \gamma'$.

3. PROGRAMMING LANGUAGE SPECIFICATIONS

Programming languages can be specified using ML in a natural way: ML signatures include the syntax of the programming language and the additional operations and predicates needed to define semantical ingredients; ML models give interpretations to the primitive data types like integers, booleans, and syntactically the language constructs; the operational semantics of the statements is given by a set of pairs of ML formulas specifying a transition relation.

Definition 5 (Programming Language Specification, PLS). A language definition is a triple $\mathcal{L} = (\Phi, \mathcal{T}, \mathcal{S})$, where

- Φ is an ML signature $(\Sigma, \Pi, \mathit{Cfg})$ giving syntax to the language's execution infrastructure (called *configuration*). *Cfg* is the sort for configurations.
- A Φ-Model $\mathcal{T}$. Recall that $\mathcal{T}_s$ denote the elements of the model $\mathcal{T}$ that have the sort s, in particular, the elements of $\mathcal{T}_{\mathit{Cfg}}$ are called *configurations*.
- A set $\mathcal{S}$ of pairs $\varphi \Rightarrow \varphi'$ of ML formulas, defining the operational semantics of the language.

A pair $\varphi \Rightarrow \varphi'$ of matching logic formulae φ and φ' is also called *reachability rule.*

Example 5. The set $\mathcal{S}(\mathrm{WH})$ of the rules that describe the operational semantics of WH is that given in Fig. 2. The operation $[\![\_]\!](\_) : \texttt{Exp}\ \mathit{State} \to \mathit{Val}$ is the evaluation of a given expression in a given state, where `Exp ::= AExp | BExp` and $\mathit{Val} = \mathbb{Z} \cup \mathbb{B}$. This function is equationally specified by structural induction on the definition of expressions. A configuration is represented as a pair $\langle\_\rangle\langle\_\rangle$, where the first component is the code to be executed and the second one is the current state. The first rule specifies the semantics of the assignment: the expression E from the right-hand side is evaluated in the current state σ and the obtained result will be the new value of the variable X in the next state. The second and the third rules specify the semantics of the `while` statement: if the evaluation of the condition E in the current state is *false*, then the execution of the statement terminates; otherwise the execution is the same with that of the `while` body followed by the same statement `while`. The last rule specifies the semantics for the block statement and is straightforward. We further consider the empty program $\cdot$ that plays the role of right neutral element for the sequential composition, *i.e.* $S\ \cdot = S$. This allows the rules to be applied to programs consisting of a single statement.

Definition 6 (Transition System Defined by a PLS). Let $\mathcal{L} = (\Phi, \mathit{Cfg}, \mathcal{T}, \mathcal{S})$ be a programming language specification. The *transition system* defined

$$\begin{array}{ll}
[\![V]\!](\sigma){=}V & [\![E_1/E_2]\!](\sigma){=}[\![E_1]\!](\sigma)/_{\mathrm{Int}}[\![E_2]\!](\sigma) \text{ if } [\![E_2]\!](\sigma){\neq}_{\mathrm{Int}}0 \\
[\![X]\!](\sigma){=}\sigma[X] & [\![E_1{<}E_2]\!](\sigma){=}[\![E_1]\!](\sigma) <_{\mathrm{Int}} [\![E_2]\!](\sigma) \\
[\![E_1{+}E_2]\!](\sigma){=}[\![E_1]\!](\sigma) +_{\mathrm{Int}} [\![E_2]\!](\sigma) & [\![E_1{==}E_2]\!](\sigma){=}[\![E_1]\!](\sigma) ==_{\mathrm{Int}} [\![E_2]\!](\sigma)
\end{array}$$

$$\begin{array}{lll}
\langle X = E;\ S\rangle\langle\sigma\rangle & \Rightarrow & \langle S\rangle\langle\sigma[X \leftarrow [\![E]\!](\sigma)]\rangle \\
\langle \texttt{while}(E)\ S\ S'\rangle\langle\sigma\rangle \wedge [\![E]\!](\sigma) ==_{\mathrm{Bool}} \mathit{false} & \Rightarrow & \langle S'\rangle\langle\sigma\rangle \\
\langle \texttt{while}(E)\ S\ S'\rangle\langle\sigma\rangle \wedge [\![E]\!](\sigma) ==_{\mathrm{Bool}} \mathit{true} & \Rightarrow & \langle S\ \texttt{while}(E)\ S\ S'\rangle\langle\sigma\rangle \\
\langle \{\ S\ \}\ S'\rangle\langle\sigma\rangle & \Rightarrow & \langle S\ S'\rangle\langle\sigma\rangle
\end{array}$$

Fig. 2. The semantics of WH.

by $\mathcal{L}$ is $(M_{Cfg}, \rightarrow_{\mathcal{S}})$, where $\rightarrow_{\mathcal{S}} = \{(\gamma,\gamma') \mid (\exists\varphi \Rightarrow \varphi' \in \mathcal{S})(\exists\rho)(\gamma,\rho) \models \varphi \wedge (\gamma',\rho) \models \varphi'\}$. We write $\gamma \rightarrow_{\mathcal{S}} \gamma'$ for $(\gamma,\gamma') \in \rightarrow_{\mathcal{S}}$. An *execution path* is a (possibly infinite) sequence of transitions

$$\tau \triangleq \gamma_0 \rightarrow_{\mathcal{S}} \gamma_1 \rightarrow_{\mathcal{S}} \cdots$$

A finite execution path is *complete* iff it is not a strict prefix of another execution path.

Example 6. Here is an example of finite execution path:

$$\begin{array}{ll}
\langle \texttt{while}(\texttt{x} < 7)\{\texttt{y} = \texttt{y} + \texttt{x}; \texttt{x} = \texttt{x}{+}1;\}\rangle\langle x \mapsto 6, y \mapsto 14\rangle & \rightarrow_{\mathcal{S}} \\
\langle \{\texttt{y}{=}\texttt{y}{+}\texttt{x}; \texttt{x}{=}\texttt{x} + 1;\}\ \texttt{while}(\texttt{x}{<}7)\{\texttt{y}{=}\texttt{y}{+}\texttt{x}; \texttt{x}{=}\texttt{x}{+}1;\}\rangle\langle x \mapsto 6, y \mapsto 14\rangle & \rightarrow_{\mathcal{S}} \\
\langle \texttt{y}{=}\texttt{y}{+}\texttt{x}; \texttt{x}{=}\texttt{x}{+}1;\ \texttt{while}(\texttt{x}{<}7)\{\texttt{y}{=}\texttt{y}{+}\texttt{x}; \texttt{x}{=}\texttt{x}{+}1;\}\rangle\langle x \mapsto 6, y \mapsto 14\rangle & \rightarrow_{\mathcal{S}} \\
\langle \texttt{x} = \texttt{x} + 1;\ \texttt{while}(\texttt{x} < 7)\{\texttt{y} = \texttt{y} + \texttt{x}; \texttt{x} = \texttt{x} + 1;\}\rangle\langle x \mapsto 6, y \mapsto 20\rangle & \rightarrow_{\mathcal{S}} \\
\langle \texttt{while}(\texttt{x} < 7)\{\texttt{y} = \texttt{y} + \texttt{x}; \texttt{x} = \texttt{x} + 1;\}\rangle\langle x \mapsto 7, y \mapsto 14\rangle & \rightarrow_{\mathcal{S}} \\
\langle\cdot\rangle\langle x \mapsto 7, y \mapsto 14\rangle &
\end{array}$$

Remark 1. The set of execution paths can be specified as the largest set defined by the following set of rules:

$$\frac{}{\gamma} \qquad \frac{\tau}{\gamma_0 \rightarrow_{\mathcal{S}} \tau}\ (\exists\varphi \Rightarrow \varphi' \in \mathcal{S})(\exists\rho)(\gamma,\rho) \models \varphi \wedge (hd(\tau),\rho) \models \varphi'$$

where hd is coinductively defined by

$$hd(\gamma) = \gamma \qquad\qquad hd(\gamma_0 \rightarrow^{\mathcal{S}}_{\tau}) = \gamma_0$$

4. ALL-PATH REACHABILITY LOGIC

All-Path Reachability Logic (APRL) is a logic suitable to express reachability properties of programs written in non-deterministic (*e.g.* concurrent) languages. There is also One-Path Reachability Logic [14] only for deterministic languages, but this is not included in this paper. In this section, we present a deductive system for APRL, deductive system that is parametric in the PLS of the language.

Definition 7. An **all-path reachability rule** is a pair $\varphi \Rightarrow^{\forall} \varphi'$ of matching logic formulae φ and φ'.

Step∃ :

$$\frac{\begin{array}{l}\models \varphi \to \exists \mathit{FreeVars}(\mathit{left}).\mathit{left} \\ \models \exists c\ (\varphi[c/\square] \wedge \mathit{left}[c/\square]) \wedge \mathit{right} \to \varphi'\end{array} \quad \text{for some } \mathit{left} \Rightarrow \mathit{right} \in \mathcal{S}}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{\exists} \varphi'}$$

Step∀ :

$$\frac{\begin{array}{l}\models \varphi \to \bigvee_{\mathit{left} \Rightarrow \mathit{right} \in \mathcal{S}} \exists \mathit{FreeVars}(\mathit{left}).\mathit{left} \\ \models \exists c\ (\varphi[c/\square] \wedge \mathit{left}[c/\square]) \wedge \mathit{right} \to \varphi' \quad \text{for each } \mathit{left} \Rightarrow \mathit{right} \in \mathcal{S}\end{array}}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{\forall} \varphi'}$$

Axiom :

$$\frac{\varphi \Rightarrow^{Q} \varphi' \in \mathcal{A}}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'}$$

Transitivity :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi_2 \qquad \mathcal{S}, \mathcal{A} \cup \mathcal{C} \vdash \varphi_2 \Rightarrow^{Q} \varphi_3}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi_3}$$

Reflexivity :

$$\frac{\cdot}{\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{Q} \varphi}$$

Consequence :

$$\frac{\models \varphi_1 \to {\varphi_1}' \qquad \mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} {\varphi_1}' \Rightarrow^{Q} {\varphi_2}' \qquad \models {\varphi_2}' \to \varphi_2}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi_2}$$

Circularity :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow^{Q} \varphi'\}} \varphi \Rightarrow^{Q} \varphi'}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'}$$

Case Analysis :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{Q} \varphi \qquad \mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow^{Q} \varphi}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \vee \varphi_2 \Rightarrow^{Q} \varphi}$$

Abstraction :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \exists \mathsf{X}\ \varphi \Rightarrow^{Q} \varphi'}$$

Fig. 3. Proof system for reachability. We make the standard assumption that the free variables of *left* ⇒ *right* in the *Step* proof rule are fresh (in particular disjoint from those of $\varphi \Rrightarrow \varphi'$). The variable Q ranges over quantifiers ∀ and ∃.

The intuitive semantics of such a rule is that any configuration γ matching φ advances, in one or more steps, into a configuration γ' matching φ' *along any execution path* that starts in γ, assuming that γ is a configuration which terminates.

In order to have a uniform notation, we use the notation $\varphi \Rightarrow^{\exists} \varphi'$ for the usual reachability rules $\varphi \Rightarrow \varphi'$, whose semantics is intuitively existential: if a configuration γ matches φ and γ terminates, then there exists an execution path starting in γ which reaches a configuration γ' matching φ'. We call such reachability formulae *one-path* reachability formulae.

The following definition captures the semantics of all-path reachability rules:

Definition 8. Assuming that $\mathcal{S}$ is a set of reachability rules in a PLS $\mathcal{L} = (\Phi, \mathit{Cfg}, \mathcal{T}, \mathcal{S})$, we write $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ iff for all complete $\rightarrow_{\mathcal{S}}$-paths τ starting with $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ and for all $\rho : \mathit{Var} \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$, there exists some $\gamma' \in \tau$ such that $(\gamma', \rho) \models \varphi'$.

We have shown [20] that there exists a sound and relatively complete proof system (given in Fig. 3) which derives all-path and one-path reachability rules from a PLS.

The system derives more general sequents of the form $\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'$, where $Q \in \{\forall, \exists\}$ denotes the type of reachability rule derived, $\mathcal{C}$ is a set of all-path and one-path reachability rules called circularities and $\mathcal{A}$ is a set of reachability rules which can be used as axioms. Circularities are similar to axioms, but their use is guarded: they can be used only after a step has been performed in order to guarantee soundness.

Circularities are introduced by the *Circularity* rule. This rule allows us to prove all repetitive constructs such as loops or recursive calls. When $\mathcal{C}$ is empty, we write $\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{Q} \varphi'$ instead of $\mathcal{S}, \mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow^{Q} \varphi'$. The circularities are "unleashed" as axioms in the *Transitivity* rule, after at least one step is taken from φ_1 to φ_2.

As a simple example, we consider the following program (which we abbreviate `SUM`):

```
while (n > 0) {
  s = s + n;
  n = n - 1;
}
```

The proof system allows us to derive the following sequent:

(1)

$$\exists s \,.\, \mathcal{S}, \emptyset \vdash \langle \texttt{SUM} \rangle \langle \mathsf{s} \mapsto s, \mathsf{n} \mapsto n \rangle \wedge n \geq 0 \Rightarrow^{\forall} \langle \cdot \rangle \langle \mathsf{s} \mapsto s + n(n+1)/2, \mathsf{n} \mapsto 0 \rangle$$

which captures the partial correctness property of the `SUM` program. Recall that $\cdot$ denotes the empty program. In proving the sequent, it is necessary to use the following circularity, which captures the invariant of the program:

(2)
$$\exists(s,n').(\langle \texttt{SUM ~> } \mathit{somecode}\rangle\langle \texttt{s}\mapsto s+(n-n')(n+n'+1)/2, \texttt{n}\mapsto n'\rangle)\wedge n'\geq 0$$
$$\Rightarrow^{\forall} \langle \mathit{somecode}\rangle\langle \texttt{s}\mapsto s+n(n+1)/2, \texttt{n}\mapsto 0\rangle.$$

The *CaseAnalysis* rule is used to distinguish between the termination of the `while` instruction and the unrolling of its loop. The fact that (2) is preserved by the `while` loop is deduced using the rules *Axiom*, *Step*, and *Transitivity*. Due to the *Circularity* rule, (2) can be deduced by unrolling just once the `while` loop. It is easy to see that (1) can be obtained from (2) by applying the *Consequence* rule.

Note that all-path reachability rules encode invariants, pre-conditions and post-conditions in a uniform way captured by circularities.

5. SYMBOLIC EXECUTION AND CIRCULAR COINDUCTION

Symbolic execution is a static program analysis technique introduced in 1976 by James C. King [9]. It consists in executing programs with symbolic inputs instead of concrete ones, and involves the processing of expressions containing symbolic values [12]. The symbolic execution can be expressed very naturally in our framework. A program configuration where the variables have symbolic values, *i.e.* a symbolic configuration, is just a particular ML formula. Hence the symbolic configuration together with the path condition can be expressed as the conjunction of ML formulas. It follows that a symbolic execution step $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$ shows how a new ML formula φ', specifying the next configurations, is derived from the formula φ, specifying the current configuration, by applying the rules from the language definition.

Definition 9 (Derivatives for ML and RL Formulas). If φ is an ML formula then
$$\Delta_{\mathcal{S}}(\varphi) \triangleq \{(\exists var(\varphi_l,\varphi_r))(\varphi_l\wedge\varphi)^{=?}\wedge\varphi_r \mid \varphi_l\Rightarrow\varphi_r\in\mathcal{S}\}$$
where $\varphi^{=?}$ is the FOL formula $(\exists z)\varphi'$ with φ' obtained from φ by replacing each basic pattern occurrence π with $z=\pi$, and with z a variable that does not occur in φ. If $\varphi\Rightarrow\varphi'$ is an RL formula then
$$\Delta_{\mathcal{S}}(\varphi\Rightarrow\varphi') \triangleq \{\varphi_1\Rightarrow\varphi' \mid \varphi_1\in\Delta_{\mathcal{S}}(\varphi)\}.$$
An ML formula φ is $\mathcal{S}$*-derivable* if $\Delta_{\mathcal{S}}(\varphi)$ is satisfiable. An RL formula $\varphi\Rightarrow\varphi'$ is $\mathcal{S}$-derivable if φ is $\mathcal{S}$-derivable.

The symbolic execution paths can be coinductively defined using the derivatives in a similar way to the coinductive specifications of the concrete execution paths.

Definition 10 (Symbolic Execution Path). The *set of symbolic execution paths* is coinductively defined by the following set of rules:

$$\frac{}{\varphi}\ \varphi \text{ satisfiable} \qquad \frac{\tau^{\mathfrak{s}}}{\varphi_0 \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \tau^{\mathfrak{s}}}\ hd(\tau^{\mathfrak{s}}) \in \Delta_{\mathcal{S}}(\varphi_0) \wedge hd(\tau^{\mathfrak{s}}) \text{ derivable}$$

where *hd* is coinductively defined by

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

The symbolic execution thus defined is related with concrete execution via coverage and precision properties [2,10]. Briefly, this means that the transition system generated by symbolic execution forward-simulates the one generated by concrete execution, and that the transition system generated by concrete execution backward-simulates the one generated by symbolic execution (restricted to satisfiable patterns). Moreover, we may automatically obtain a language definition $\mathcal{L}^{\mathfrak{s}}$ whose concrete executions are symbolic executions of $\mathcal{L}$ [2,18]. The symbolic execution can be used to prove reachability formulas with all-path semantics using the following proof system consisting only of two rules:

Definition 11 (SYSTEP).

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

Let $\nu\mathsf{SYSTEP}$ denote the largest set defined by the system SYSTEP (see [10] for details.)

Definition 12 (Totality). A set $\mathcal{S}$ of RL formulas is *total* iff for each $\mathcal{S}$-derivable φ and each (γ, ρ) such that $(\gamma, \rho) \models \varphi$, there is γ_1 such that $\gamma \rightarrow_{\mathcal{S}} \gamma_1$.

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

THEOREM 1. *If $\mathcal{S}$ is total then $\mathcal{S} \models \nu\mathsf{SYSTEP}$.*

Specifically, if one can construct a finite proof tree under SYSTEP for a given RL formula then the formula belongs to $\nu\mathsf{SYSTEP}$. Theorem 1 then says that the formula holds in the semantics $\mathcal{S}$. An RL formula also holds if a infinite proof tree under SYSTEP can be built. We give below an example of an infinite proof tree, which we reuse in order to show how *circular coinduction* can "fold" infinite proof trees into finite ones in a stronger proof system.

Example 7. A proof tree for

$$\langle \texttt{while(x!=0)\{s=s+x; x=x-1;\}}\ S\rangle\langle \texttt{x} \mapsto a\ \texttt{s} \mapsto 0\rangle \Rightarrow$$
$$(\exists b)\langle S\rangle\langle \texttt{x} \mapsto 0\ \texttt{s} \mapsto b\rangle \wedge b =_{Int} \frac{a(a +_{Int} 1)}{2}$$

is represented in Fig. 4. T_1 corresponds to the case when the **`while`** condition is false. One can see that T_2 is infinite and corresponds to the infinitely many unfoldings of the **`while`** loop.

$$T_2$$

$$[\text{der}]\ \frac{[\text{der}]\ \frac{[\text{der}]\ \frac{\cdots}{\begin{array}{l}\begin{array}{l}\langle\mathtt{while\,(x\ !=\ 0)\,\{s=s+x;\ x=x-1;\}}\ S\rangle\\ \langle \mathtt{x}\mapsto a -_{Int} 1\ \mathtt{s}\mapsto a\rangle\end{array} \wedge a \neq_{Int} 0\\ \Rightarrow\\ (\exists b)\langle S\rangle\langle \mathtt{x}\mapsto 0\ \mathtt{s}\mapsto b\rangle \wedge b =_{Int} \dfrac{a(a+_{Int}1)}{2}\end{array}}}{\begin{array}{l}\begin{array}{l}\langle\mathtt{x=x-1;while\,(x\ !=\ 0)\,\{s=s+x;\ x=x-1;\}}\ S\rangle\\ \langle \mathtt{x}\mapsto a\ \mathtt{s}\mapsto a\rangle\end{array} \wedge a \neq_{Int} 0\\ \Rightarrow\\ (\exists b)\langle S\rangle\langle \mathtt{x}\mapsto 0\ \mathtt{s}\mapsto b\rangle \wedge b =_{Int} \dfrac{a(a+_{Int}1)}{2}\end{array}}}{\begin{array}{l}\begin{array}{l}\langle\mathtt{s=s+x;\ x=x-1;while\,(x\ !=\ 0)\,\{s=s+x;\ x=x-1;\}}\ S\rangle\\ \langle \mathtt{x}\mapsto a\ \mathtt{s}\mapsto 0\rangle\end{array} \wedge a \neq_{Int} 0\\ \Rightarrow\\ (\exists b)\langle S\rangle\langle \mathtt{x}\mapsto 0\ \mathtt{s}\mapsto b\rangle \wedge b =_{Int} \dfrac{a(a+_{Int}1)}{2}\end{array}}$$

$$T_1$$

$$[\text{impl}]\ \frac{}{\begin{array}{l}\langle S\rangle\langle \mathtt{x}\mapsto a\ \mathtt{s}\mapsto 0\rangle \wedge a =_{Int} 0 \Rightarrow\\ \quad (\exists b)\langle S\rangle\langle \mathtt{x}\mapsto 0\ \mathtt{s}\mapsto b\rangle \wedge b =_{Int} \dfrac{a(a+_{Int}1)}{2}\end{array}}$$

$$\frac{T_1 \qquad T_2}{\begin{array}{l}\langle\mathtt{while\,(x!=0)\,\{s=s+x;\ x=x-1;\}}\ S\rangle\langle \mathtt{x}\mapsto a\ \mathtt{s}\mapsto 0\rangle \Rightarrow\\ (\exists b)\langle S\rangle\langle \mathtt{x}\mapsto 0\ \mathtt{s}\mapsto b\rangle \wedge b =_{Int} \dfrac{a(a+_{Int}1)}{2}\end{array}}$$

Fig. 4. An infinite proof tree under SYSTEP.

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

Definition 13 (Symbolic Circular Coinduction). Let G be a finite set of $\mathcal{S}$-derivable RL formulas. Then the set of rules $\mathsf{SCC}(G)$ is SYSTEP together with

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

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

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

Theorem 2 (Circularity Principle). *Assume $\mathcal{S}$ total and that for each $\varphi_c \Rightarrow \varphi'_c \in G$, $var(\varphi'_c) \subseteq var(\varphi_c)$. If $\Delta_{\mathcal{S}}(G) \subseteq \nu\,\mathsf{SCC}(G)$ then $\mathcal{S} \models G$.*

Note that $\mathcal{S} \not\models \nu\,\mathsf{SCC}(G)$ in general. For instance, for any arbitrary set of RL formulas G, each $\varphi \Rightarrow \varphi' \in G$ is in $\nu\,\mathsf{SCC}(G)$ by applying the rule [circ]. Theorem 2 identifies a subset of proof trees under $\mathsf{SCC}(G)$ that are sound w.r.t. $\mathcal{S} \models \_$ (a proof tree for $\varphi \Rightarrow \varphi'$ under $\mathsf{SCC}(G)$ is sound w.r.t. $\mathcal{S} \models \_$ if $\mathcal{S} \models \varphi \Rightarrow \varphi'$): those for which the root is derived using the rule [der].

$$
[\text{der}]\ \dfrac{\begin{array}{c}[\text{circ}]\ \dfrac{[\text{impl}]\ \dfrac{}{\begin{array}{l}(\exists b)\langle S\rangle\langle \mathtt{x} \mapsto 0\ \mathtt{s} \mapsto b\rangle \wedge b =_{Int} \dfrac{(a -_{Int} 1)a}{2} +_{Int} (s_0 +_{Int} a)\\ \Rightarrow\\ (\exists b)\langle S\rangle\langle \mathtt{x} \mapsto 0\ \mathtt{s} \mapsto b\rangle \wedge b =_{Int} \dfrac{a(a +_{Int} 1)}{2} +_{Int} s_0\end{array}}}{\begin{array}{l}\begin{array}{l}\langle \mathtt{while\,(x!=0)\,\{s=s+x;\ x=x-1;\}}\ S\rangle\\ \langle \mathtt{x} \mapsto a -_{Int} 1\ \mathtt{s} \mapsto s_0 +_{Int} a\rangle\end{array} \wedge a \neq_{Int} 0\\ \Rightarrow\\ (\exists b)\langle S\rangle\langle \mathtt{x} \mapsto 0\ \mathtt{s} \mapsto b\rangle \wedge b =_{Int} \dfrac{a(a +_{Int} 1)}{2} +_{Int} s_0\end{array}}\\ \hline \\ \cdots\end{array}}{\begin{array}{l}\begin{array}{l}\langle \mathtt{s=s+x;\ x=x-1;while\,(x!=0)\,\{s=s+x;\ x=x-1;\}}\ S\rangle\\ \langle \mathtt{x} \mapsto a\ \mathtt{s} \mapsto s_0\rangle\end{array} \wedge a \neq_{Int} 0\\ \Rightarrow\\ (\exists b)\langle S\rangle\langle \mathtt{x} \mapsto 0\ \mathtt{s} \mapsto b\rangle \wedge b =_{Int} \dfrac{a(a +_{Int} 1)}{2} +_{Int} s_0\end{array}}
$$

Fig. 5. The finite proof tree under SCC corresponding to the infinite proof tree T_2 in Fig. 4.

Example 8. The finite proof tree under SCC that corresponds to the infinite proof tree T_2 under SYSTEP is represented in Fig. 5.

6. PROGRAM EQUIVALENCE

Can matching logic be used to reason about program equivalence? In this section, we summarize work showing that it can. We first show that, given two (deterministic) programming languages specified using matching logic, their definitions can be aggregated into a single definition such that programs in the resulting language are pairs of programs from the initial languages. Fig. 6 shows how the syntax of the two languages can be aggregated.

$$\begin{tikzcd}
(S_0, \Sigma_0) \arrow[r, "h_2"] \arrow[d, "h_1"'] & (S_2, \Sigma_2) \arrow[d, "h'_2"] \\
(S_1, \Sigma_1) \arrow[r, "h'_1"] & (S', \Sigma')
\end{tikzcd}$$

Fig. 6. Pushout diagram for the syntax of two programming languages.

The pushout theorem states that if the two signatures (S_1, Σ_1) and (S_2, Σ_2) have some commonalities identified by the signature (S_0, Σ_0), then the aggregated signature (S', Σ') exists and is unique up to certain assumptions. In the aggregated signature, it is possible to express programs from both programming languages.

By the amalgamation theorem, if models of (S_1, Σ_1) and (S_2, Σ_2) exist which agree on (S_0, Σ_0), then a model of the aggregated syntax exists as well. In the aggregated signature, we add an additional constructor $\langle \_, \_ \rangle$ for aggregated configurations. Fig. 7 summarizes this construction.

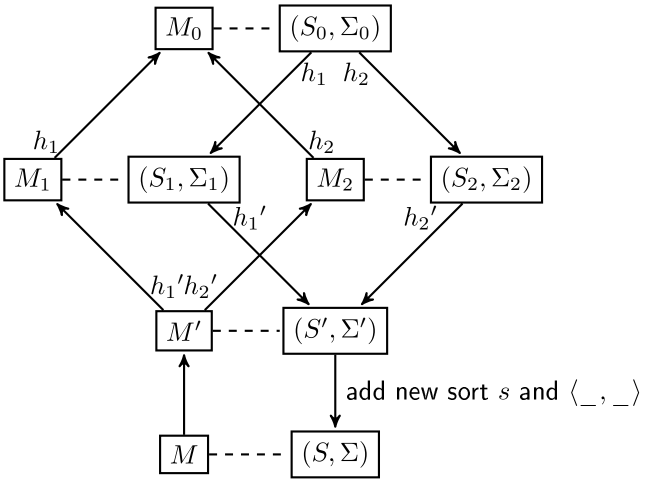

Fig. 7. Construction of the aggregated model M of (S, Σ).

Once the aggregated language is constructed, it can be used to prove various equivalences between programs. Two programs are partially equivalent if, for all inputs on which they both terminate, they produce the same output. Two programs are fully equivalent if they terminate on exactly the same set of inputs and furthermore they produce the same results.

Depending on how the reachability rules of the aggregated language are specified, it is possible to reduce constructively partial equivalence of two programs to partial correctness [6] in the aggregated language or it is possible to prove full equivalence [5] using a dedicated proof system shown in Fig. 8.

$$\textsc{Axiom}\ \frac{\varphi \in E}{\vdash \varphi \Downarrow^{\infty} E} \qquad \textsc{Conseq}\ \frac{\models \varphi \to \exists \tilde{x}.\varphi' \qquad \vdash \varphi' \Downarrow^{\infty} E}{\vdash \varphi \Downarrow^{\infty} E}$$

$$\textsc{Case Analysis}\ \frac{\vdash \varphi \Downarrow^{\infty} E \qquad \vdash \varphi' \Downarrow^{\infty} E}{\vdash \varphi \vee \varphi' \Downarrow^{\infty} E}$$

$$\textsc{Step}\ \frac{\models \varphi_1 \Rightarrow_1^{*} {\varphi_1}' \qquad \models \varphi_2 \Rightarrow_2^{*} {\varphi_2}' \qquad \vdash \langle {\varphi_1}', {\varphi_2}' \rangle \Downarrow^{\infty} E}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^{\infty} E}$$

$$\textsc{Circularity}\ \frac{\models \varphi_1 \Rightarrow_1^{+} {\varphi_1}' \quad \models \varphi_2 \Rightarrow_2^{+} {\varphi_2}' \quad \vdash \langle {\varphi_1}', {\varphi_2}' \rangle \Downarrow^{\infty} E \cup \{\langle \varphi_1, \varphi_2 \rangle\}}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^{\infty} E}$$

Fig. 8. Full Equivalence Proof System.

The main proof rule of the system is Circularity, which allows to postulate synchronization points in the two programs. The proof system has been used [5] to prove that two programs computing the Collatz sequence are equivalent, even if it is not known whether they terminate or not.

7. THE K FRAMEWORK

$\mathbb{K}$ (`http://kframework.org` [15, 17]) is a RL-based semantic framework for the definition of programming languages, type systems, and analysis tools. In addition to paradigmatic programming languages that are used for teaching, several real-life programming languages have been completely defined in $\mathbb{K}$, including C11 [7, 8], Java 1.4 [4], and Javascript ES5 [11].

The $\mathbb{K}$ syntax for programming languages is a BNF-like notation enriched with (optional) semantic annotations attached to syntactic productions. For example, the syntax of arithmetic expressions in WH (shown in Fig. 1) can be given in $\mathbb{K}$ as follows:

```
syntax AExp ::= Int
              | Id
              | AExp "/" AExp   [left, strict]
              > AExp "+" AExp   [left, strict]
              | (AExp)          [bracket]
```

The meta-connective > is used to express the fact that division has higher precedence than addition. Besides the annotations needed for parsing purposes (*e.g.* `left` – for left associativity, `bracket` – for removing the production from the parse tree), the semantic annotation `strict` is used to instruct the 𝕂 tool that the arguments of the corresponding syntactical construct (*e.g.*, the arguments of `+` and `/`) must be evaluated before the entire construct is evaluated. Under the hood, this is done by generating the so-called *heating/cooling* rules. This allows to write the evaluation of each arithmetic operator by a single rule:

```
I1 + I2                    => I1 +Int I2
I1 / I2 => I1 /Int I2 requires I2 =/=Int 0,
```

where `+Int`, `/Int`, and `=/=Int` are the addition, division, and dis-equality test, respectively, in the model. `I1` and `I2` are variables of sort `Int`. The rule for division is equivalent to the ML formula

```
I1 / I2 ∧ I2 =/=Int 0 => I1 /Int I2
```

In 𝕂, configurations are nested structures of cells; for our simple language WH, the 𝕂 configuration is similar with the one used in Example 2:

`<k>` *computations* `</k>` `<state>` *state* `</state>`

The cell labelled `<k>` holds of the code to be executed represented as a sequence of computation tasks, while the `<state>` cell holds the state represented as finite set of mappings from program variables to their values. An example of concrete configuration is

```
<k> 5 + 2 ~> x = HOLE; ~> y = x / 2; </k>
<state> a -> 5 x |-> -2 y |-> 0 </state>|,
```

where the content of the cell `k` can be read as: compute first `5 + 2` (using semantic rules), then put the result instead of the special variable `HOLE` (using cooling rules), then compute the assignment `x = 5;` (using the below rule), after which compute the assignment `y = x / 2;` in the same manner. The content of the cell `state` is self-contained.

The semantics of a language in 𝕂 can be defined using 𝕂 rules, which are reachability rules, where the left-hand side is a topmost ML formula (*i.e.*, a conjunction between a configuration term and a side condition) and the right-hand side is a configuration term (which is a ML formula as well). The 𝕂 rules make it explicit which parts of the configurations are locally changed and use an abstraction mechanism that allows to specify the minimal context needed for doing the changes. To have a hint how the $\mathbb{K}$ rules look, we consider the rule for assignment:

```
<k> X = I:Int; => . ...</k>
<state> Sigma:Map => Sigma[I/X] </state>
```

The above rule actually says the followings: 1) the assignment `X = I`, where `I` is an integer, is the first computation task in the `<k>` cell and in the next configuration it is replaced with the empty computation `.` (the rest of the cell remains unchanged), and 2) the current state `Sigma` is replaced in the next configuration with the new one where the value of `X` is updated with `I` and in the rest. To understand the abstraction mechanism, we consider the rule sequential composition

```
S1 S2 => S1 ~> S2
```

that is equivalent to

```
<k> S1 S2 => S1 ~> S2 ...</k>
<state> Sigma:Map </state>
```

and it actually says that "execute first `S1` and then `S2`. As we can already see, the $\mathbb{K}$ definition of while is simpler than that given in Fig. 2.

For further details about the features of the $\mathbb{K}$ framework and how to define languages using $\mathbb{K}$ the reader is referred to [15, 19] and invited to follow the tutorial (`http://www.kframework.org/index.php/K_Tutorial`).

The $\mathbb{K}$ framework provides a compiler for language definitions and a runner for programs. The definition compiler generates an interpreter for the language in question, while the runner calls the interpreter over the program given as argument, and outputs the final configuration. Below we show how to compile the $\mathbb{K}$ definition of WH (stored in a file `wh.k`) and how to run the WH program `SUM` given in Section 4, where the variable `n` is initialized with 10:

```
-$ kompile wh.k
-$ krun sum.wh
<k> .K </k> <state> s |-> 55 n |-> 0 </state>
```

The basic features of $\mathbb{K}$ together with the advanced ones are presented in detail in [19].

In addition to defining the semantics of programming languages, $\mathbb{K}$ can also be used for program analysis and verification. There were already defined or are in progress a prototype for one-path reachability logic used to verify C-like programs [16], an extension of the framework that allows to perform symbolic execution using directly the 𝕂 definition of a language [18], a circular-coinduction-based prototype that implements the SCC proof system [1], and a pushdown model-checker based on [3]. Currently, there is ongoing work on standardisation of 𝕂 definitions, on developing of a specialised symbolic rewrite engine, and on its integration with various provers.

8. CONCLUSION

The programming languages must have a standard formal semantics. Now, the programming language manuals (standards) include only an appendix, where only the grammar of the syntax is given. We strongly believe that the same thing should happen with the semantics. We showed that the Matching-Logic-based approach supplies a theoretical foundation for giving semantics for programming languages and to use this semantics for program execution, verification, and analysis. Within the 𝕂 Framework project we showed that this approach is practical and suitable for real-life programming languages like C and Java.

REFERENCES

[1] A. Arusoaie, D. Lucanu and V. Rusu, *A Generic Framework for Symbolic Execution.* Research Report, RR-8189, Inria, Sept. 2015. Available at `https://hal.inria.fr/hal-00766220`.

[2] A. Arusoaie, D. Lucanu and V. Rusu, *Symbolic execution based on language transformation.* Computer Languages, Systems & Structures **44** (2015), 48–71.

[3] I.M. Asăvoae, F.S. de Boer, M.M. Bonsangue, D. Lucanu and Jurriaan Rot, *Model checking recursive programs interacting via the heap.* Sci. Comput. Program. **100** (2015), 61–83.

[4] D. Bogdănaş and G. Roşu, *K-Java: A Complete Semantics of Java.* Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15), ACM, January 2015, 445-456.

[5] Ş. Ciobâcă, D. Lucanu, V. Rusu and G. Roşu, *A language-independent proof system for full program equivalence.* Form. Asp. Comput. 2016, 1–29.

[6] Ş. Ciobâcă, *Reducing Partial Equivalence to Partial Correctness.* SYNASC 2014, 164–171, IEEE, 2014.

[7] C. Ellison and G. Roşu, *An Executable Formal Semantics of C with Applications.* Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12), ACM, January 2012, 533–544.

[8] C. Hathhorn, C. Ellison and G. Roşu, *Defining the Undefinedness of C.* Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15), ACM, June 2015, 336–345.

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

[10] D. Lucanu, V. Rusu and A. Arusoaie, *A Generic Framework for Symbolic Execution: Theory and Applications.* J. Symbolic Comput. To appear.

[11] D. Park, A. Ştefănescu and G. Roşu, *KJS: A Complete Formal Semantics of JavaScript.* Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15), ACM, June 2015, pp. 346–356.

[12] C.S. Păsăreanu and Willem Visser, *A survey of new trends in symbolic execution for software testing and analysis.* International Journal on Software Tools for Technology Transfer **11** (2009), *4*, 339-353.

[13] G. Roşu, *Matching Logic — Extended Abstract.* Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA'15) **36**, *Leibniz International Proceedings in Informatics (LIPIcs).* Dagstuhl, Germany, July 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 5–21.

[14] G. Roşu, A. Ştefănescu, Ş. Ciobâcă and B.M. Moore, *One-Path Reachability Logic.* LICS 2013, IEEE 2013, 358-367.

[15] G. Roşu and T.-F. Şerbănuţă, *K Overview and SIMPLE Case Study.* Proceedings of International K Workshop (K'11). ENTCS **304**, June 2014, pp. 3-56.

[16] G. Roşu and A. Ştefănescu, *Checking Reachability using Matching Logic.* OOPSLA, ACM 2012, pp. 555-574.

[17] G. Roşu and T.-F. Şerbănuţă, *An overview of the K semantic framework.* J. Log. Algebr. Program. **79** (2010), *6*, 397-434.

[18] V. Rusu, D. Lucanu, T.-F. Şerbănuţă, A. Arusoaie, A. Ştefănescu and G. Roşu, *Language definitions as rewrite theories.* J. Log. Algebr. Meth. Program. **85** (2016), *1*, 98–120.

[19] T-F Şerbănuţă, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu and G. Roşu, *The Primer (version 3.3).* Electron. Notes Theor. Comput. Sci. **304** (2014), 57–80, Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[20] A. Ştefănescu, Ş. Ciobâcă, R. Mereuţă, B.M. Moore, T.-F. Şerbănuţă and G. Roşu, *All-Path Reachability Logic.* RTA-TLCA'14. LNCS **8560**, July 2014, pp. 425-440.

Received 1 August 2016

"Alexandru Ioan Cuza" University,
Romania
stefan.ciobaca@info.uaic.ro
dlucanu@info.uaic.ro

University of Illinois
Urbana-Champaign, USA
grosu@illinois.edu

Inria Lille, France
vlad.rusu@inria.fr

University of Bucharest
traian.serbanuta@gmail.com