Language Definitions as Rewrite Theories

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu, Traian-Florin Serbanuta, Andrei Stefanescu, Grigore Rosu

Language Definitions as Rewrite Theories

Andrei Arusoaie[1], Dorel Lucanu[1], Vlad Rusu[2], Traian-Florin Şerbănuţă[1,3],
Andrei Ştefănescu[4], and Grigore Roşu[4]

[1] Alexandru Ioan Cuza University, Iaşi, Romania
[2] Inria Lille Nord Europe, France
[3] University of Bucharest, Romania
[4] University of Illinois at Urbana-Champaign, USA

Abstract. $\mathbb{K}$ is a formal framework for defining the operational semantics of programming languages. It includes software tools for compiling $\mathbb{K}$ language definitions to Maude rewrite theories, for executing programs in the defined languages based on the Maude rewriting engine, and for analyzing programs by adapting various Maude analysis tools. A recent extension to the $\mathbb{K}$ tool suite is an automatic transformation of language definitions that enables the symbolic execution of programs, i.e., the execution of programs with symbolic inputs. In this paper we investigate the theoretical relationships between $\mathbb{K}$ language definitions and their translations to Maude, between symbolic extensions of $\mathbb{K}$ definitions and their Maude encodings, and how the relations between $\mathbb{K}$ definitions and their symbolic extensions are reflected on their respective representations in Maude. These results show, in particular, how analyses performed with Maude tools can be formally lifted up to the original language definitions.

1 Introduction

$\mathbb{K}$ [11] is a framework for formally defining the semantics of programming languages. The current version of $\mathbb{K}$ includes options that have Maude [3] as a backend: the $\mathbb{K}$ compiler transforms any $\mathbb{K}$ definition into a Maude module; then, the $\mathbb{K}$ runner uses Maude to run or analyze programs in the defined language.

Recently, $\mathbb{K}$ has been extended with symbolic execution support [2]. Briefly, a $\mathbb{K}$ language definition is automatically transformed into a *symbolic-language* definition, such that the concrete executions of programs using the symbolic definition are symbolic executions of programs using the original language definition. The transformation amounts to incorporating *path conditions* in program configurations, and to changing the language's semantic rules so that they match on *symbolic configurations* and that they automatically update the path conditions.

Symbolic executions are called *feasible* if their path conditions are satisfiable. Two results relating concrete and symbolic program executions are proved in [2]: *coverage*, saying that for each concrete execution there is a feasible symbolic one taking the same path on the program; and *precision*, saying that for each feasible symbolic execution there is a concrete one taking the same program path.

In this paper we propose two ways of representing $\mathbb{K}$ language definitions in Maude: a *faithful* representation and an *approximate* one. We then study the relationships between $\mathbb{K}$ language definitions (including the symbolic ones, obtained by the above-described transformation) and their representations in Maude. We also show how the coverage and precision results, which relate a language $\mathcal{L}$ and its symbolic extension $\mathcal{L}^{\mathfrak{s}}$, are reflected on their respective representations in Maude. These results show, in particular, how (symbolic) analyses performed with Maude tools on the (faithful and approximate) Maude representations of languages can be lifted up to the original language definitions. The various results that we have obtained can be graphically depicted as in following diagram (dashed arrows show the results proved in the paper):

Faithful Representation

$$\begin{tikzcd}
\mathcal{L} \arrow[r, leftrightarrow, "\text{coverage}", "\text{precision}"'] \arrow[d, dashed, "\text{faithful encoding}"'] & \mathcal{L}^{\mathfrak{s}} \arrow[d, dashed, "\text{faithful encoding}"] \\
\mathcal{R}(\mathcal{L}) \arrow[r, leftrightarrow, dashed, "\text{coverage}", "\text{precision}"'] & \mathcal{R}(\mathcal{L}^{\mathfrak{s}})
\end{tikzcd}$$

Approximate Representation

$$\begin{tikzcd}
\mathcal{L} \arrow[r, leftrightarrow, "\text{coverage}", "\text{precision}"'] \arrow[d, dashed, "\text{approx. encoding}"'] & \mathcal{L}^{\mathfrak{s}} \arrow[d, dashed, "\text{approx. encoding}"] \\
\mathfrak{R}(\mathcal{L}) \arrow[r, dashed, "\text{coverage}"'] & \mathfrak{R}(\mathcal{L}^{\mathfrak{s}}) \arrow[ul, dashed, "\text{precision}"]
\end{tikzcd}$$

In the faithful encoding, each semantic rule of the language definition $\mathcal{L}$ is translated into a rewrite rule of the rewrite theory $\mathcal{R}(\mathcal{L})$. Equations are only introduced in order to express equality in the data domain. The resulting rewrite theory is proved to be *executable* by Maude, and the transition system generated by the language definition is shown to be isomorphic to the one generated by the rewrite theory. Some variations of this encoding are also discussed, all of which satisfy the executability and faithfulness properties. As a consequence, both positive and negative results of reachability analyses, obtained on rewrite theories (i.e., by using the Maude *search* command) also hold on the original language definitions. Moreover, all symbolic reachability analysis results obtained on the rewrite-theory representation $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ of a symbolic language $\mathcal{L}^{\mathfrak{s}}$ also hold on the rewrite-theory representation $\mathcal{R}(\mathcal{L})$ of the language $\mathcal{L}$. The latter property is analoguous to the results obtained in [10], where *rewriting modulo SMT* is shown to be related to (usual) rewriting in a *sound* and *complete* way.

For nontrivial language definitions, the faithful encoding is not very practical, because it typically generates a huge state-space that is not amenable to reachability analysis. This is why we introduce approximate representations of language definitions as *two-layered rewrite theories*. These approximations are obtained by splitting the semantic rules of the language into two sets, called *layers*, such that the first layer forms a terminating rewrite system. The one-step rewriting in such a theory is obtained by computing an irreducible form w.r.t. rules from the first layer (according to a given strategy), and then applying a rule from the second layer. A simple example of a two-layered rewrite theory is a Maude module consisting of equations and rules, where the equations (denoting the first layer) are only required to be terminating, and both the equations and rules (which form the second layer) specify transitions in the underlying transition-system model of the theory.

In an (approximating) two-layered rewrite theory $\mathfrak{R}(\mathcal{L})$, only a subset of the executions of programs in the original language $\mathcal{L}$ are represented. The consequence is that only positive results of reachability analyses on the two-layered rewrite theories can be lifted up to the corresponding language definitions. In addition to reducing the state-space to be explored, the approximate encoding of a language by a two-layered rewrite theory can also be seen as the output of a *compiler* that solves some semantic choices left by the language definition at compile-time. For example, in C, the order in which the operands of addition are evaluated is a compile-time choice. By turning the operand-evaluation rules into first-layer rules, and by letting Maude automatically execute these rules in various orders according to certain strategies, one can reproduce the various design compile-time choices for the evaluation of arguments.

We note that approximating two-layered rewrite theories have some limitations: only the coverage property relating the language definition $\mathcal{L}$ to its symbolic version $\mathcal{L}^{\mathfrak{s}}$ also holds on their respective approximate encodings theories; the precision property holds only in some restricted cases. However, the precision property between the approximate symbolic encoding $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ and the language definition $\mathcal{L}$ always holds. Hence, one can trace symbolic reachability analyses (performed on $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$) back to programs in $\mathcal{L}$, and also (in some restricted cases) to the representation of programs in $\mathfrak{R}(\mathcal{L})$, which, as discussed above, can be seen as compiled programs where some semantic choices are left to the compiler.

Organisation. In Section 2 we present our working examples, which are two programs belonging to the CinK kernel of C++, which was specified in $\mathbb{K}$ [7]. A partial description of the $\mathbb{K}$ definition for CinK is included. In Section 3 we introduce a formal notion of a language-definition framework, which allows us to make our approach independent of the $\mathbb{K}$ language definitional framework and to abstract away some particular implementation details of $\mathbb{K}$. For the same reason, we will be using rewrite theories (instead of their implementations as Maude modules) for the encodings of language definitions. We also briefly present the language-independent symbolic execution approach [2] and recap some essential notions related to the executability of rewrite theories.

Section 4 presents the faithful and the approximate representations of language definitions into a rewrite theory and the various relations between them (graphically depicted in the above diagram). Section 5 presents the applications of these representations to the compilation of $\mathbb{K}$ language definitions as Maude modules. Finally, Section 6 presents conclusions and related work.

2 Running Example

Our running example is CinK [7], a kernel of the C++ programming language. The $\mathbb{K}$ definition of CinK can be found on the $\mathbb{K}$ Framework Github repository: `http://github.com/kframework/cink-semantics`. As any $\mathbb{K}$ definition, it consists of the language syntax, given using a BNF-style grammar, and of its semantics, given using rewrite rules on configurations. In this paper we only exhibit a small part of the $\mathbb{K}$ definition of CinK, whose syntax is shown in Figure 1. Some of the grammar productions are annotated with $\mathbb{K}$-specific attributes.

A major feature of C++ expressions is that given by the “sequenced before” relation [1], which defines a partial order over the evaluation of subexpressions.

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

Fig. 1. CinK syntax

This can be easily expressed in $\mathbb{K}$ using the *strict* attribute to specify an evaluation order for an operation's operands. If the operator is annotated with the *strict* attribute then its operands will be evaluated in a nondeterministic order. For instance, all the binary operations are strict. Hence, they may induce non-determinism in programs because of possible side-effects in their arguments.

Another feature is given by the classification of expressions into *rvalues* and *lvalues*. The arguments of binary operations are evaluated as rvalues and their results are also rvalues, while, e.g., both the argument of the prefix-increment operation and its result are lvalues. The *strict* attribute for such operations has a sub-attribute *context* for wrapping any subexpression that must be evaluated as an rvalue. Other attributes ($\mathit{funcall}, \mathit{divide}, \mathit{plus}, \mathit{minus}, \dots$) are names associated to each syntactic production, which can be used for referring to them.
The $\mathbb{K}$ framework uses *configurations* to store program states. A configuration is a nested structure of cells, which typically include the program to be executed, input and output streams, values for program variables, and other additional information. The configuration of CinK (Figure 2) includes the $\langle\rangle_{\mathsf{k}}$ cell containing the code that remains to be executed, which is represented as a list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \dots$ to be executed in the given order. Computation tasks are typically statements and expression evaluations. The memory is modeled using two cells $\langle\rangle_{\mathsf{env}}$ (which holds a map from variables to addresses) and $\langle\rangle_{\mathsf{state}}$ (which holds a map from addresses to values). The configuration also includes a cell for the function call stack and another one for the return values of functions.

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

Fig. 2. CinK configuration

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

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

$I_1{:}Int\ \texttt{+}\ I_2{:}Int \Rightarrow I_1 +_{Int} I_2$ [*plus*]
$I_1{:}Int\ \texttt{/}\ I_2{:}Int \Rightarrow I_1 /_{Int} I_2$ **requires** $I_2 \neq_{Int} 0$ [*division*]
$\texttt{if(}\ true\ \texttt{)}\ St{:}Stmt\ \texttt{else}\ \_ \Rightarrow St$ [*if-true*]
$\texttt{if(}\ false\ \texttt{)}\ \_\ \texttt{else}\ St{:}Stmt \Rightarrow St$ [*if-false*]
$\texttt{while(}\ B{:}Exp\ \texttt{)}\ St{:}Stmt \Rightarrow \texttt{if(}\ B\ \texttt{)\{}\ St\ \texttt{while(}\ B\ \texttt{)}\ St\ \texttt{else \{\}\}}$ [*while*]
$V{:}Val\ \texttt{;} \Rightarrow \cdot$ [*instr-expr*]
$\langle \texttt{++lval(}\ L{:}Loc\ \texttt{)} \Rightarrow \texttt{lval(}\ L\ \texttt{)}\ \cdots\rangle_{\mathsf{k}} \langle \cdots\ L \mapsto (V{:}Int \Rightarrow V +_{Int} 1\ \cdots\rangle_{\mathsf{store}}$ [*inc*, *memw*]
$\langle \texttt{--lval(}\ L{:}Loc\ \texttt{)} \Rightarrow \texttt{lval(}\ L\ \texttt{)}\ \cdots\rangle_{\mathsf{k}} \langle \cdots\ L \mapsto (V{:}Int \Rightarrow V -_{Int} 1\ \cdots\rangle_{\mathsf{store}}$ [*dec*, *memw*]
$\langle\langle \texttt{lval(}\ L{:}Loc\ \texttt{)=}\ V{:}Val \Rightarrow V\ \cdots\rangle_{\mathsf{k}} \langle\cdots\ L \mapsto \_ \Rightarrow V\ \cdots\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathsf{cfg}}$ [*update*, *memw*]
$\langle\langle \texttt{\$lookup(}\ L{:}Loc\ \texttt{)} \Rightarrow V\ \cdots\rangle_{\mathsf{k}}\langle\cdots\ L \mapsto V{:}Val\ \cdots\rangle_{\mathsf{store}}\ \cdots\rangle_{\mathsf{cfg}}$ [*lookup*, *memr*]
$\texttt{\{}\ Sts{:}Stmts\ \texttt{\}} \Rightarrow Sts$ [*block*]

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

where $+_{Int}$ is the mathematical operation for addition. Note that the ellipses in a cell (e.g., $\langle\ \cdots\rangle_{\mathsf{k}}$) represent the part of the cell not affected by the rule.

The rule for division has a side condition which restricts its application. The conditional statement `if` has two corresponding rules, one for each possible evaluation of the condition expression. The rule for the `while` loop is unrolled into an `if` statement. The increment and update rules have side effects in the $\langle\rangle_{\mathsf{store}}$ cell, modifying the value stored at a specific address. Finally, the reading of a value from the memory is specified by the lookup rule, which matches a value in the $\langle\rangle_{\mathsf{store}}$ and places it in the $\langle\rangle_{\mathsf{k}}$ cell. The auxiliary construct `$lookup` is used, e.g., when a program variable is evaluated as an rvalue.

In addition to these rules (writtten by the $\mathbb{K}$ user), the $\mathbb{K}$ framework automatically generates so-called *heating* and *cooling* rules, which are induced by *strict* attributes. We show only the case of division, which is strict in both arguments:

$$A_1\ \texttt{/}\ A_2 \Rightarrow rvalue(A_1) \curvearrowright \square\ \texttt{/}\ A_2 \quad (1) \qquad rvalue(I_1) \curvearrowright \square\ \texttt{/}\ A_2 \Rightarrow I_1\ \texttt{/}\ A_2 \quad (3)$$

$$A_1\ \texttt{/}\ A_2 \Rightarrow rvalue(A_2) \curvearrowright A_1\ \texttt{/}\ \square \quad (2) \qquad rvalue(I_2) \curvearrowright A_1\ \texttt{/}\ \square \Rightarrow A_1\ \texttt{/}\ I_2 \quad (4)$$

where $\square$ is a special symbol, destined to receive the result of an evaluation.

We shall be using the following two programs in the sequel. The program `counter` in Figure 4 is nondeterministic; nondeterminism arises from the unde-

```
int counter = 1;
int inc() {
  return ++counter;
}
int dec() {
  return --counter;
}
int main() {
  return  inc() + dec();
}
```

a) The program `counter`

```
int main() {
int k, x;
  x = A:Int;  //A:Int is a symbolic value
  k = 0;
  while (x > 0) {
    ++k;
    x = x / 2;
  }
}
```

b) The program `log`

Fig. 4. Two C++ programs

fined evaluation order for the arguments of the `+` operation and from the side-effects in its arguments. The program `log` in the same figure is a symbolic one because `A:Int` is a symbolic value, which can denote any integer value. When it is completed the variable `k` holds $[\log_2(A)]$ where $[\_]$ denotes the integer part of a real number. In Section 5 we show how the behaviours of these programs can be analysed using our encodings of the CinK language as Maude programs.

3 Background

3.1 The Ingredients of a Language Definition

In this section we identify the ingredients of language definitions in an algebraic and term-rewriting setting. The concepts are explained on the $\mathbb{K}$ definition of CinK. We assume the reader is familiar with the basics of algebraic specification and rewriting. A language $\mathcal{L}$ can be defined as a triple $(\Sigma, \mathcal{T}, \mathcal{S})$, consisting of:

1. A many-sorted algebraic signature Σ, which includes at least a sort *Cfg* for *configurations* and a sort *Bool* for *constraint formulas*. For the sake of presentation, 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. 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 infinite set of variables *Var*, let $T_\Sigma(\mathit{Var})$ denote the free Σ-algebra of terms with variables, $T_{\Sigma,s}(\mathit{Var})$ denote the set of terms of sort s with variables, and $\mathit{var}(t)$ denote the set of variables occurring in the term t.
2. A Σ^{Data}-model $\mathcal{D}$, which interprets the data sorts and operations. For convenience, we assume that $\mathcal{D}_d \subset \Sigma_d$ for each data sort d, i.e., the constants are elements of the corresponding signature. Let $\mathcal{T} \triangleq \mathcal{T}(\mathcal{D})$ denote the free Σ-model generated by $\mathcal{D}$. The satisfaction relation $\rho \models b$ between valuations ρ and constraint formulas $b \in T_{\Sigma,Bool}(\mathit{Var})$ is defined by $\rho \models b$ iff $\rho(b) = \mathcal{D}_{true}$. For simplicity, we write $\mathit{true}, \mathit{false}, 0, 1 \ldots$ instead of $\mathcal{D}_{true}, \mathcal{D}_{false}, \mathcal{D}_0, \mathcal{D}_1, \ldots$.
3. A set $\mathcal{S}$ of rewrite rules. Each rule is a pair of the form $l \wedge b \Rightarrow r$, where $l, r \in T_{\Sigma,\mathit{Cfg}}(\mathit{Var})$ are the rule's *left-hand-side* and *right-hand-side*, respectively, and $b \in T_{\Sigma,Bool}(\mathit{Var})$ is the *condition*. The formal definitions for rules and for the transition system defined by them are given below.

Remark 1. For the sake of presentation, here we consider only "pure" language definitions, where the semantics is given only by semantic rules between configurations. Some definitions may include additional functions defined by equations. For such cases the language definition may additionally includes a set of axioms A_0, e.g., associativity and/or commutativity of some functions, and a set of equations E_0. Then the model $\mathcal{T}$ is the free algebra modulo $A_0 \cup E_0$. We believe that the approach presented in this paper can be extended to these more involved definitions, but this requires more investigation and is left for future work.

We now formally introduce the notions required for defining semantic rules.

Definition 1 (pattern [12]). *A* pattern *is an expression of the form $\pi \wedge b$, where $\pi \in T_{\Sigma,Cfg}(Var)$ is a* basic pattern *and $b \in T_{\Sigma,Bool}(Var)$. If $\gamma \in T_{Cfg}$ and $\rho : Var \to \mathcal{T}$ then we write $(\gamma, \rho) \models \pi \wedge b$ iff $\gamma = \rho(\pi)$ and $\rho \models b$.*

A basic pattern π defines a set of (concrete) configurations, and the condition b gives additional constraints these configurations must satisfy.

Remark 2. The above definition is a particular case of a definition in [12]. There, a pattern is a first-order logic formula with configuration terms as sub-formulas. In this paper we keep the conjunction notation from first-order logic but separate basic patterns from constraints. Note that first-order formulas can be encoded as terms of sort *Bool*, where the quantifiers become constructors. The satisfaction relation $\models$ is then defined, for such terms, like the usual FOL satisfaction.

We identify basic patterns π with patterns $\pi \wedge \mathit{true}$. Sample patterns are $\langle\langle I_1 \texttt{ + } I_2 \curvearrowright C\rangle_{\mathsf{k}} \langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ and $\langle\langle I_1 \texttt{ / } I_2 \curvearrowright C\rangle_{\mathsf{k}} \langle Env\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$.

Definition 2 (rule, transition system). *A* rule *is a pair of patterns of the form $l \wedge b \Rightarrow r$ (note that r is in fact the pattern $r \wedge \mathit{true}$). Any set $\mathcal{S}$ of rules defines a labelled transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$ such that $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff there exist $\alpha \triangleq (l \wedge b \Rightarrow r) \in \mathcal{S}$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models l \wedge b$ and $(\gamma', \rho) \models r$.*

3.2 Symbolic Execution

We briefly recap our approach to symbolic execution from [2]. The main idea is to automatically generate a new definition $(\Sigma^{\mathsf{s}}, \mathcal{T}^{\mathsf{s}}, \mathcal{S}^{\mathsf{s}})$ for a language $\mathcal{L}^{\mathsf{s}}$ 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.

Let V^{s} denote an infinite, data sort-wise set of *symbolic values*, disjoint from *Var* and from symbols in Σ. The data algebra is extended to $\mathcal{D}^{\mathsf{s}}$, which is the algebra of ground terms over the signature $\Sigma^{\mathsf{Data}}(V^{\mathsf{s}})$.

Remark 3. The approach in [2] allows some freedom in choosing the algebra $\mathcal{D}^{\mathsf{s}}$, to enable the use of decision procedures for handling symbolic artifacts.

The signature Σ^{s} extends Σ with the symbolic values V^{s} as constants, a new sort Cfg^{s} and a constructor $\_ \wedge \_ : Cfg \times Bool \to Cfg^{\mathsf{s}}$. The model $\mathcal{T}^{\mathsf{s}}$ is defined as being the free Σ^{s}-model generated by $\mathcal{D}^{\mathsf{s}}$, similarly to how $\mathcal{T}$ is built over $\mathcal{D}$. The ground terms $\pi \wedge \phi \in \mathcal{T}^{\mathsf{s}}_{Cfg^{\mathsf{s}}}$ are called *symbolic configurations*. Let $[\![\pi \wedge \phi]\!]$ denote the set of concrete configurations $\{\gamma \mid (\exists\rho)\, (\gamma, \rho) \models \pi \wedge \phi\}$.

Thanks to the rule transformation procedure presented in [2], we make without loss of generality the assumption that the basic patterns in left-hand sides of rules do not contain operations on data, and the rules are left-linear. Concrete semantic rules $l \wedge b \Rightarrow r \in \mathcal{S}$ are then systematically transformed into rules

$$l \wedge \psi \Rightarrow r \wedge (\psi \wedge b) \tag{5}$$

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

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 $\mathcal{S}^{\mathsf{s}}$. The transition system $(\mathcal{T}^{\mathsf{s}}_{\mathit{Cfg}^{\mathsf{s}}}, \Rightarrow_{\mathcal{S}^{\mathsf{s}}})$ is defined using Definitions 1, 2 applied to $\mathcal{L}^{\mathsf{s}}$. In [2] it is proved 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.

Theorem 1 (Coverage [2]). *For every concrete execution* $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}} \cdots$ *there is a symbolic execution* $\pi_0 \wedge \phi_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \pi_1 \wedge \phi_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \pi_n \wedge \phi_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

A symbolic configuration $\pi \wedge \phi \in \mathcal{T}^{\mathsf{s}}_{\mathit{Cfg}^{\mathsf{s}}}$ is *satisfiable* if there is a valuation $\vartheta : V^{\mathsf{s}} \to \mathcal{D}$ such that $\vartheta \models \phi$ (which is equivalent to $[\![\pi \wedge \phi]\!] \neq \emptyset$). We call a symbolic execution *feasible* if all its configurations are satisfiable.

Theorem 2 (Precision [2]). *For every feasible symbolic execution* $\pi_0 \wedge \phi_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \pi_1 \wedge \phi_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \pi_n \wedge \phi_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}^{\mathsf{s}}} \cdots$ *there is a concrete execution* $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n \overset{\alpha_{n+1}}{\Longrightarrow}_{\mathcal{S}} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

3.3 Rewrite Theories

A rewrite theory [3] $\mathcal{R} = (\Sigma, E \cup A, R)$ consists of a signature Σ, a set of equations E, a set of axioms A, e.g., associativity, commutativity, unity or combinations of these, and a set of rewrite rules R of the form $l \to r$ **if** b, where l and r are terms with variables and b is a term of sort *Bool*. We are only interested in rewrite theories $\mathcal{R}$ that are *executable*, i.e., $(\Sigma, E \cup A, R)$ where:

1. there exists a matching algorithm modulo A;
2. $(\Sigma, E \cup A)$ is ground Church-Rosser and terminating modulo A (the equations E are seen here as rewrite rules oriented from left to right). Thus, each ground term t has a canonical form $\mathit{can}_{E/A}(t)$ that is unique modulo the axioms A;
3. R is *ground coherent w.r.t. E modulo A* [13]: for all $t, t_1 \in T_\Sigma$ with $t \to_{R/A} t_1$ there is $t_2 \in T_\Sigma$ s.t. $\mathit{can}_{E/A}(t) \to_{R/A} t_2$ and $\mathit{can}_{E/A}(t_1) =_A \mathit{can}_{E/A}(t_2)$.

The relation $\to_{R/A}$ denotes the one-step rewriting relation defined by applying a rule from R modulo axioms A: $u \to_{R/A} v$ iff there are the terms u', v', a rule $l \to r$ **if** b in R, position p in u', and substitution σ such that $u =_A u'$, $v =_A v'$, $u'|_p = \sigma(l)$[5], $v' = u[\sigma(r)]_p$[6], and $\sigma(b) =_A \mathit{true}$.

The *rewriting relation* $\to_{\mathcal{R}}$ defined by an executable rewrite theory $\mathcal{R}$ is: $t_1 \to_{\mathcal{R}} t_2$ iff $\mathit{can}_{E/A}(t_1) \to_{R/A} t_2'$ and $\mathit{can}_{E/A}(t_2') = t_2$. This is equivalent to $\to_{R/(E \cup A)}$ due to confluence and coherence. We write $t_1 \overset{\alpha}{\to}_{\mathcal{R}} t_2$ to emphasise that $\alpha \triangleq (l \to r$ **if** $b) \in R$ is applied in the rewriting step $\mathit{can}_{E/A}(t_1) \to_{R/A} t_2'$.

[5] $t|_p$ denotes subterm of t at position p.

[6] $t[u]_p$ denotes the term obtained from t by replacing the subterm at position p with u.

4 Translating Language Definitions into Rewrite Theories

This section includes the main contribution of the paper. We introduce two encodings of language definitions as rewrite theories: a faithful encoding and an approximate encoding. Since the symbolic extension of a language is also a language definition, we automatically get encodings of both concrete languages and their symbolic extensions. We investigate how the properties relating a language definition and its symbolic extension are reflected on their respective encodings.

Definition 3 (faithful encoding). *Let $\mathcal{L} = (\varSigma, \mathcal{T}, \mathcal{S})$ be a language definition. The* faithful encoding *of $\mathcal{L}$ is $\mathcal{R}(\mathcal{L}) = (\varSigma, E \cup A, R)$, where*

- *$A = \emptyset$;*
- *for each operation f in $\varSigma^{\mathsf{Data}}$ and $d_1, \ldots, d_n \in \mathcal{D}$ of corresponding sorts, E includes an equation $f(d_1, \ldots, d_n) = \mathcal{D}_f(d_1, \ldots, d_n)$;*
- *$R = \mathcal{S}$, where each rule $\pi \wedge b \Rightarrow r \in \mathcal{S}$ becomes a rewrite rule $l \to r$ **if**$b \in R$.*

Theorem 3. *Let $\mathcal{L} = (\varSigma, \mathcal{T}, \mathcal{S})$ be a language definition. Then $\mathcal{R}(\mathcal{L})$ is an executable rewrite theory satisfying $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff $\gamma \xrightarrow{\alpha}_{\mathcal{R}(\mathcal{L})} \gamma'$, for all $\gamma, \gamma' \in \mathcal{T}_{Cfg}$.*

Remark 4. The construction of the rewrite theory $\mathcal{R}(\mathcal{L})$, with data domain $\mathcal{D} \subseteq \varSigma^{\mathsf{Data}}$ defined by the set of equations E given in Definition 3, corresponds to the data domains $\mathcal{D}$ being *builtin sorts* in the Maude terminology. A builtin sort is a sort that is not built algebraically but one that, for efficiency reasons, is directly implemented in code (C++ code in the case of Maude). For example, natural numbers are specified by the equational specification $0 : \mathsf{Nat}, s : \mathsf{Nat} \to \mathsf{Nat}$, but using the resulting unary-notation for them would be highly inefficient. This is why natural numbers are implemented as builtins. The construction $\mathcal{R}(\mathcal{L})$ can, however, be extended to accomodate non-builtin sorts, i.e., sorts that are defined as the *initial model* of a finite set of equations E' that are confluent and terminating modulo a set A of axioms. For this, it is enough to ensure that $E' \cup E$ is also confluent and terminating modulo A - where E is the set of equations given in the proof of Theorem 3. This typically happens, as E and E' refer to different sorts - the builtin ones for the former, and the non-builtin ones for the latter. If this is the case then the proof of the ground coherence property in Theorem 3 still holds, because it only depends on $E' \cup E$ being confluent and terminating modulo A, not on the particular form of the equations. The proof of faithfulness of the encoding remains the same. This observation is important, since it ensures that we obtain executable Maude rewrite-theories $\mathcal{R}(\mathcal{L})$ for languages-definitions $\mathcal{L}$ whose data are specified using either bulitin sorts or non-builtin sorts. The faithfulness of the encoding then ensures that all results of reachability analyses (either positive or negative) performed on $\mathcal{R}(\mathcal{L})$, e.g., obtained using Maude's *search* command, also hold on $\mathcal{L}$.

The symbolic extension of a language definition can be encoded as a rewrite theory as well. Let $\mathcal{L}^{\mathfrak{s}} = (\varSigma^{\mathfrak{s}}, \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ be the symbolic extension of $\mathcal{L} = (\varSigma, \mathcal{T}, \mathcal{S})$. Recall that $\varSigma^{\mathfrak{s}}$ is $\varSigma$ extended with the constructor of symbolic configurations

$\_ \wedge \_$ and with the symbolic values $V^{\mathfrak{s}}$ seen as constants. The symbolic configurations are ground terms $\pi \wedge \phi \in \mathcal{T}^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}$. If $\mathcal{R}(\mathcal{L}^{\mathfrak{s}}) = (\Sigma^{\mathfrak{s}}, E \cup A, R)$ is the faithful encoding given by Theorem 3, then $E = A = \emptyset$ because the data algebra $\mathcal{D}^{\mathfrak{s}}$ we considered is the $\Sigma^{\mathsf{Data}}(V^{\mathfrak{s}})$-algebra of the ground terms built over $\mathcal{D}$ and $V^{\mathfrak{s}}$. Recall that we assumed that $\mathcal{D} \subseteq \Sigma \subseteq \Sigma^{\mathsf{Data}}(V^{\mathfrak{s}})$.

The relationship between a language definition $\mathcal{L}$ and its symbolic extension $\mathcal{L}^{\mathfrak{s}}$ can be now reflected at the level of the encodings $\mathcal{R}(\mathcal{L})$ and $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$. A symbolic configuration $\pi \wedge \phi$ consists of a configuration ground term π (of sort *Cfg*) and a formula ground term ϕ (of sort *Bool*). The constants $V^{\mathfrak{s}}$ play the role of logical variables, and the definition of satisfiability for patterns extends to their representations as symbolic configurations. Moreover, the notion of feasible execution in $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ is defined similarly to how it is defined for $\mathcal{L}^{\mathfrak{s}}$. The following two results are direct consequences of Theorems 3, 1, and 2, respectively.

Corollary 1 (Coverage for Encoding Rewrite Theories). *For every concrete execution* $\gamma_0 \xrightarrow{\alpha_0}_{\mathcal{R}(\mathcal{L})} \gamma_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \gamma_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L})} \cdots$ *there is a symbolic execution* $\pi_0 \wedge \phi_0 \xrightarrow{\alpha_1}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \pi_n \wedge \phi_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

Corollary 2 (Precision for Encoding Rewrite Theories). *For every feasible symbolic execution* $\pi_0 \wedge \phi_0 \xrightarrow{\alpha_1}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \pi_n \wedge \phi_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ *there is a concrete execution* $\gamma_0 \xrightarrow{\alpha_0}_{\mathcal{R}(\mathcal{L})} \gamma_1 \xrightarrow{\alpha_2}_{\mathcal{R}(\mathcal{L})} \cdots \xrightarrow{\alpha_n}_{\mathcal{R}(\mathcal{L})} \gamma_n \xrightarrow{\alpha_{n+1}}_{\mathcal{R}(\mathcal{L})} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$.

The faithful encoding thus enjoys nice theoretical properties, but it has a limited practical value when we consider actual $\mathbb{K}$ definitions of nontrivial languages:

- The heating and cooling rules, which are symmetric each other, may lead to infinite rewritings;
- The generated state space may be very large, even for small programs.

There are currently two proposals for obtaining abstractions of the rewrite theories: equational abstraction [9] or transforming some semantical rules into equations [6].

The former amounts to basically deriving a new definition, where the new model $\mathcal{T}$ is the quotient of the the original one, usually requiring substantial input from the user, which is something we would like to avoid.

The latter might not be suitable for language definitions in general because, semantically, it would equate elements that are supposed to be distinct in $\mathcal{T}$. Consider a language construct `randBool` with two rules: `randBool => true` and `randBool => false`. Assume now we want to analyze a program which uses `randBool`, but who fails to satisfy a given property regardless of whether `randBool` transits to `true` or to `false`. In this case it might beneficial to collapse the state space by considering only one of the cases; however, if we transform the two rules above into equations, this will semantically identify `true` and `false` in $\mathcal{T}$, collapsing much more of the state space than desirable. An additional operational concern is that transforming certain rules into equations might destroy coherence and/or confluence, thus falling out of the executability requirements.

Two-layered rewrite theories, introduced below, allow us to preserve the benefits of the techniques above (state space reduction, efficient execution), while avoiding their semantical consequences (unnecessary collapse of states in the semantical model $\mathcal{T}$).

Definition 4. *A* two-layered rewrite theory *is a tuple* $\mathfrak{R} = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$*, where* $(\Sigma, E \cup A, 1R \cup 2R)$ *is an executable rewrite theory,* $E \cup 1R$ *is ground terminating modulo* A*, and* $\varepsilon : T_\Sigma \to T_\Sigma$ *is a function that, for any* $t \in T_\Sigma$*, returns an element in the set of* $(E \cup 1R)/A$*-irreducible terms* $\{t' \in T_\Sigma \mid t \to^{!}_{(E \cup 1R)/A} t'\}$ *(which is nonempty precisely because* $E \cup 1R$ *is ground terminating modulo* A*). The one-step rewrite relation* $\twoheadrightarrow_{\mathcal{R}}$ *is defined by* $t_1 \twoheadrightarrow_{\mathcal{R}} t_2$ *iff* $\varepsilon(t_1) \to_{2R/A} t'_2$ *and* $can_{E/A}(t'_2) =_A t_2$*.*

Theorem 4. *Let* $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ *be a language definition and* $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ *be a two-layered rewrite theory with* $(\Sigma, E \cup A, 1R \cup 2R)$ *built as in Definition 3 but where the set of rules is partitioned into two subsets 1R and 2R and* $E \cup 1R$ *is terminating modulo* A*. If* $\gamma \twoheadrightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$ *then* $\gamma \Rightarrow^{+}_{\mathcal{S}} \gamma'$*.*
We say that $\mathfrak{R}(\mathcal{L})$ *is an* approximate encoding *of* $\mathcal{L}$*.*

Corollary 3 (precision for approximate encoding). *Let* $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ *be a language definition and* $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ *be an approximate encoding of* $\mathcal{L}^{\mathfrak{s}}$*. For each feasible symbolic execution* $\pi_0 \wedge \phi_0 \twoheadrightarrow_{\mathcal{R}^{\mathfrak{s}}} \pi_1 \wedge \phi_1 \twoheadrightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \twoheadrightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \twoheadrightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ *there is a concrete execution in* $\mathcal{L}$*:* $\gamma_0 \overset{\alpha_1}{\Longrightarrow}{}^{+}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}{}^{+}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}{}^{+}_{\mathcal{S}} \gamma_n \overset{\alpha_{n+1}}{\Longrightarrow}{}^{+}_{\mathcal{S}} \cdots$ *such that* $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ *for* $i = 0, 1, \ldots$*.*

An interesting and practically relevant question is whether the coverage/precision relationships between $\mathcal{L}$ and $\mathcal{L}^{\mathfrak{s}}$ can be reflected on the level of the approximate encodings as two-layered rewrite theories. To investigate these relationships, we have to find a way to define an approximate two-layered rewrite theory $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ that extends a given approximate two-layered rewrite theory $\mathcal{R}(\mathcal{L})$. A first attempt is to define $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}}) = (\Sigma^{\mathfrak{s}}, E \cup A, 1R^{\mathfrak{s}} \cup 2R^{\mathfrak{s}}, \varepsilon^{\mathfrak{s}})$ from $\mathcal{R}(\mathcal{L})$ in the same way $\mathcal{L}^{\mathfrak{s}}$ is obtained from $\mathcal{L}$, but this is not enough to have a coverage-like result. The program `log` in Figure 4 is deterministic and terminating for each $\vartheta(A) \in Int$. So we may execute any instance of it with an approximate encoding $\mathcal{R}$ having no second-layer rules, i.e., $2R = \emptyset$. If $2R^{\mathfrak{s}} = \emptyset$, then $1R^{\mathfrak{s}}$ is non terminating because there is an infinite execution corresponding to the case when the value of the program variable `X` in the current configuration is always greater the zero. Another problem is to specify how the strategy ε is extended to $\varepsilon^{\mathfrak{s}}$. Since it is hard to give general definitions for these questions, we opted for a particular solution that can be implemented in Maude.

Definition 5 (symbolic approximate encoding). *Let* $\mathcal{L}^{\mathfrak{s}} = (\Sigma^{\mathfrak{s}}, \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ *be the symbolic extension of* $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ *and* $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, 1R \cup 2R, \varepsilon)$ *an approximate encoding of* $\mathcal{L}$*. We assume that there is a total order relation* $\prec$ *over 1R such that:*

1. *the rewrite* $t \to^{!}_{(E \cup 1R)/A} \varepsilon(t)$ *uses the minimal rule from 1R w.r.t.* $\prec$ *whenever such a rule is applicable;*

2. if α is unconditional and α' is conditional then $\alpha \prec \alpha'$.

We let the approximated encoding of $\mathcal{L}^{\mathfrak{s}}$ be $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}}) = (\Sigma^{\mathfrak{s}}, E \cup A, \mathit{1R}^{\mathfrak{s}} \cup \mathit{2R}^{\mathfrak{s}}, \varepsilon^{\mathfrak{s}})$:

- *$\mathit{1R}^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{1R}, \alpha$ unconditional$\}$;*
- *$\mathit{2R}^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{1R}, \alpha$ conditional$\} \cup \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{2R}\}$;*
- *$\alpha^{\mathfrak{s}} \prec^{\mathfrak{s}} \alpha'^{\mathfrak{s}}$ iff $\alpha \prec \alpha'$;*
- *$\varepsilon^{\mathfrak{s}}$ uses the minimal rule from $\mathit{1R}^{\mathfrak{s}}$ w.r.t. $\prec^{\mathfrak{s}}$.*

Theorem 5 (coverage for approximate rewrite theories). *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \varepsilon)$ be an approximate encoding of $\mathcal{L}$. For every concrete execution $\gamma_0 \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_1 \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_n \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots$ there is a symbolic execution $\pi_0 \wedge \phi_0 \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ such that $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ for $i = 0, 1, \ldots$.*

However, the precision relationship between $\mathfrak{R}(\mathcal{L})$ and $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ does not hold in general. The reason is that $\mathit{1R}^{\mathfrak{s}}$ has fewer rules than $\mathit{1R}$ and hence the representative-selection strategy $\varepsilon^{\mathfrak{s}}$ is weaker than ε. Therefore there are no guarantees that the concrete execution given by Corollary 3 will be the same with that chosen by the strategy ε. If the strategy $\varepsilon^{\mathfrak{s}}$ is the "isomorphic image" of ε via the transformation $\bullet \mapsto \bullet^{\mathfrak{s}}$, then the precision result holds:

Theorem 6 (precision for approximate rewrite theories). *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \varepsilon)$ be an approximated encoding of $\mathcal{L}$ such that $\mathit{1R}$ includes only unconditional rules (hence $\mathit{1R}^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{1R}\}$). For every feasible symbolic execution $\pi_0 \wedge \phi_0 \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ there is a concrete one $\gamma_0 \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_1 \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_n \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots$ such that $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ for $i = 0, 1, \ldots$.*

5 Implementing the $\mathbb{K}$ Framework in Maude

The current implementation of the $\mathbb{K}$ framework uses Maude as a rewrite engine. In [4], the framework, at that time called K-Maude, was presented as an extension of Maude consisting in several meta-transformations which gradually translate $\mathbb{K}$ modules into executable Maude modules. In the current version of $\mathbb{K}$ we use a compiler for language definitions where each of these meta-transformations is actually a separate compilation step. Through compilation, $\mathbb{K}$ definitions are translated into Maude rewrite theories which are then used for running/analysing programs. The main components of a $\mathbb{K}$ definition are the syntax declarations, the configuration and the $\mathbb{K}$ (rewrite) rules. To these, the tool adds automatically the rules generated from strictness annotations (e.g. heating/cooling rules 1-4).

The work described in this article is concerned with how the set of rules is compiled into a two-layered rewrite theory, which is then encoded into Maude by using equations for the first-layer rules and rewrite rules for the second-layer rules. By default, all $\mathbb{K}$ rules are translated into (conditional) equations, that is $\mathit{1R} = \mathcal{S}$ and $\mathit{2R} = \emptyset$. This behavior can be altered by specifying (at compile time) that certain rules are to be considered *transitions*, which will trigger their transformation into (conditional) rewrite rules in the resulted Maude module.

To specify that a rule is a transition, one must pass the rule name as an argument for the `-transition` option at compilation time:

```
$ kompile cink.k -transition "division"
```

The above command specifies the rule *division* as a transition; thus, the rule for division is included in *2R*. By this command we express our intent that the tool considers the rule for division as a transition when exploring an execution's transition system. By making it a rewrite rule in Maude, we can explore the non-determinism generated by the rule when using Maude's *search* command.

Another source of non-determinism arises from strictness annotations. When the *strict* attribute is given to some syntactical construct, the tool chooses by default an arbitrary, fixed order to evaluate its arguments. This optimisation has the side effect of possibly losing behaviours due to missed interleavings.

Some of these missed interleavings can be restored using the `-superheat` option. This option is used to instruct the $\mathbb{K}$ tool to exhaustively explore all the non-deterministic evaluation choices for the strictness of a language construct.

Once we know which rules are transitions and which are not, we can easily deduce the two sets *1R* and *2R*, and thus we obtain the executable rewrite theory $\mathfrak{R}(\mathcal{L})$ as discussed in Section 4.

The following example shows how one can explore more behaviours by specifying second-layer rules at compile time. If we compile the language definition of CinK without any options, then running the program `counter` (Figure 4) will result in a single solution, where the return value is either 1 (when the tool first evaluates `dec()` and then `inc()`) or 3 (when it first evaluates `inc()` and then `dec()`). However, if we set the operation *plus* as superheat:

```
$ kompile cink -superheat "plus"
```

then we obtain both solutions, because the heating rule for addition can be applied in two ways and the option tells the tool to explore them both.

The symbolic transformations discussed in Section 3.2 are implemented as compilation steps in the $\mathbb{K}$ compiler [2]. The tool uses the same translation to Maude discussed above in order to obtain the rewrite theory $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$. An important step in this process is that conditional rules whose conditions cannot be reduced to *true* are compiled as transitions, that is, they are included in *2R*. When performing search in Maude, these rules are essential in exploring all the execution paths, thereby ensuring the Coverage (Theorem 5) property. Note that none of the symbolic transformations applied by the tool to the language definition changes the initial semantics of the language.

The implementation uses a slightly modified version of Maude which includes a hook to the Z3 SMT solver [5] and a corresponding operation called *checkSat*. It receives as argument an SMTLib string, which is sent to the solver to check its satisfiability. The result returned by the solver is propagated back through the hook to Maude as a string, so *checkSat* can return "sat", "unsat", or "unknown". In practice, our tool uses checkSat to reduce the search space by slicing unfeasible execution paths, and thus being very important in preserving the precision property. To obtain $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ from a language definition one uses the symbolic backend as follows:

```
$ kompile cink -backend symbolic
```

This command applies the symbolic transformations, moves the appropriate rules in *2R*, and generates the rewrite theory $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$. Using $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ one can execute programs using either concrete values or symbolic ones. However, running programs with symbolic values may lead to infinite loops when the loop conditions contain symbolic values. In such cases one can bound the number of execution paths:

```
$ krun log.imp -search -bound 3 -cIN=".List" -cPC="true"
```

This executes `log` (Figure 4) symbolically, until a number of 3 solutions is found. Each solution consists in a result configuration and a formula which constitutes the path condition. The symbolic values are represented as fresh variables with a specific sort (e.g. `A:Int`). These can also be passed as input at the command line of the tool as arguments of the `-cIN` parameter. Users can also set the initial path condition using the `-cPC` option. During the symbolic execution the tool applies a rule only if the next state is feasible: the current path condition and the new conditions imposed by the application of the rule are not “unsat”.

6 Conclusion and Related Work

We presented some results that relate language definitions to different kinds of rewrite theories, which encode the language definitions both faithfully and approximately. The results show how (symbolic) analyses performed on a rewrite theory are reflected on the corresponding language definition. The general results are applied to the current implementation of $\mathbb{K}$ language definitions in Maude.

The faitfful encoding of $\mathbb{K}$ language definitions as rewrite theories is relatively simple but the resulting theory is not efficient in practice. Therefore we extended the notion of rewrite theory in order to work with under-approximations of the language definitions (and implicitly of the rewrite theories). The approximating theories are more efficient and flexible – the user has the freedom to work with various levels of approximations –, but heir use for program analysis must be done with care because they do not preserve all the behavioural properties. The coverage/precision results proved in this paper can help the user in correctly assessing which analyses hold on which representations.

Related Work $\mathbb{K}$ started as methodology for defining the semantics of the programming languages in Maude. The first tool supporting $\mathbb{K}$ [4] was written in Maude’s meta-level, as a series of transformations translating $\mathbb{K}$ definitions into Maude programs. Then the $\mathbb{K}$ compiler became a more complex tool that translates a $\mathbb{K}$ definition into an intermediate language, which is then used to generate code for various backends, including Maude. A presentation of this tool is given in [8]. There, a brief description of the semantics of $\mathbb{K}$ definitions is also included. The programming-language definition framework presented here in Section 3 is a specialised case of that definition.

The coverage and precision properties, which relate the faithful rewrite-theory encoding of a language and of that language’s symbolic version, are analoguous to the soundness and completeness results in [10], which relate usual rewriting and rewriting modulo SMT. An interesting alternative to defining symbolic execution by as executions in a transformed language (as we do it in [2]) would be to compile a language into a rewriting-modulo-SMT Maude module.

Our construction of two-layered rewrite theories have some similarities with equational abstractions [9] and with the state-space reduction techniques obtained by transforming rules into equations presented in [6]. However, our first-layer rewrite rules do not equate states as Maude equations do; their semantics is that of transformation, not of equality. Therefore these rules do not have to satisfy the executability and property-preservation requirements of [9,6].

References

1. Standard for Programming Language C++. Working Draft. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.
2. 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/`.
3. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
4. T. F. Şerbanuţă and G. Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In P. C. Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 104–122, 2010.
5. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
6. A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In *P. Proceedings of the 21st German Annual Conference on Artificial Intelligence*, pages 142–157. Springer, 2006.
7. D. Lucanu and T. F. Serbanuta. Cink - an exercise on how to think in k. Technical Report TR 12-03, Version 2, Alexandru Ioan Cuza University, Faculty of Computer Science, December 2013.
8. D. Lucanu, T. F. Şerbănuţă, and G. Roşu. The K Framework distilled. In *9th International Workshop on Rewriting Logic and its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012. Invited talk.
9. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theor. Comput. Sci.*, 403(2-3):239–264, 2008.
10. C. Rocha, J. Meseguer, and C. A. Munoz. Rewriting modulo SMT. Technical Report 218033, NASA, 2013.
11. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
12. 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.
13. P. Viry. Equational rules for rewriting logic. *Theor. Comput. Sci.*, 285(2):487–517, 2002.

This appendix is for reviewers' convenience, to see the proofs of the main results included in the paper and the output of the reported experiments. If this paper is accepted then the appendix will be removed, the proofs will be included into a technical report, and the reader will be referred to the web interface of the $\mathbb{K}$ Tool, `http://fmse.info.uaic.ro/tools/K/`, where these examples and many others can be executed online.

A Proofs

Theorem 3. *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition. Then $\mathcal{R}(\mathcal{L})$ is an executable rewrite theory satisfying $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff $\gamma \overset{\alpha}{\rightarrow}_{\mathcal{R}(\mathcal{L})} \gamma'$, for all concrete configurations $\gamma, \gamma' \in \mathcal{T}_{Cfg}$.*

Proof. In this proof we are going to construct $\mathcal{R}(\mathcal{L})$ as follows:

For the rewrite theory constructed as in Definition 3 there exists a *matching algorithm modulo the equational axioms* A, since $A = \emptyset$[7]. The equational theory $(\Sigma(\mathcal{D}), E)$ is *terminating* since each application of an equation $f(d_1, \ldots, d_n) = \mathcal{D}_f(d_1, \ldots, d_n)$ to a term strictly reduces the number of non-constant operations in Σ^{Data} in the term, by replacing non-constant terms $f(d_1, \ldots, d_n)$ by constants $\mathcal{D}_f(d_1, \ldots, d_n)$.

We now show that the equational theory (Σ, E) is also *locally confluent* (and by Newman's lemma, it is also confluent). Let t be a term and two equations $e_1 \triangleq f_1(d_1, \ldots, d_n) = \mathcal{D}_{f_1}(d_1, \ldots, d_n)$ and $e_2 \triangleq f_2(d'_1, \ldots, d'_{n'}) = \mathcal{D}_{f_2}(d'_1, \ldots, d'_{n'})$ such that they match the term t at positions p_1 and p_2, respectively. If one of the positions is a strict prefix of the other, say, p_1 is a strict prefix of p_2, then at most one of the equations can be applied (since for the other one, the arguments are not constants as required by the equation), and local confluence follows trivially.

If $p_1 = p_2$ then e_1 coincides with e_2 and thus the confluence is not an issue. If p_1 and p_2 are disjoint positions, both e_1 and e_2 can be applied, which gives:

- $t[\mathcal{D}_{f_1}(d_1, \ldots, d_n)]_{p_1}[\mathcal{D}_{f_2}(d'_1, \ldots, d'_{n'}]_{p_2}$ if e_1 is applied first, then e_2 is applied;
- $t[\mathcal{D}_{f_2}(d'_1, \ldots, d'_{n'}]_{p_2}[\mathcal{D}_{f_1}(d_1, \ldots, d_n)]_{p_1}$ if e_2 is applied first, then e_1 is applied.

Since p_1 and p_2 are disjoint, the above terms are syntactically equal, which concludes the proof of the confluence property.

We now establish the ground *coherence property.* Let t, t_1, t_2 be three ground terms, $e \triangleq f(d_1, \ldots, d_n) = \mathcal{D}_f(d_1, \ldots, d_n)$ an equation in E, $\alpha \triangleq l \rightarrow r$ **if** b a rule in R, p_e, p_α two positions in t, and σ a substitution such that $t \rightarrow_{E/A} t_1$ holds by applying e at position p_e and $t \rightarrow_{R/A} t_2$ holds by applying α at position p_α via σ (recall that $A = \emptyset$). It follows that $t_1 = t[\mathcal{D}_f(d_1, \ldots, d_n)]_{p_e}$ and $t_2 = t[\sigma(r)]_{p_\alpha}$ and $\sigma(b) \rightarrow^*_{E/A}$ *true* (b is a side condition). We will show that there are t'_1 and u such that $t_1 \rightarrow_{R/A} t'_1$, $t_2 \rightarrow^*_{E/A} u$ and $t'_1 \rightarrow^*_{E/A} u$. This will prove that the

[7] Even if $A = \emptyset$, we prefer to keep the references to the axioms A in the rest of the proof, because constructions of $\mathcal{R}(\mathcal{L})$ with $A \neq \emptyset$ also exist, cf. Remark 4.

equation does not introduce incoherence. The process can then be repeated for all the equations in order to establish the coherence property.

Since rules are applied at the top (on terms of sorts *Cfg*) and left-hand sides of equations are terms of *Data* sorts, it follows that p_α is a prefix of p_e (i.e. e is applied below α). Since l is linear and each of its data subterms is a variable, there is a unique data variable X and a data term v such that $\sigma(X) \to_{E/A} v$ holds by applying e. Let σ' be the substitution defined in the same way as σ except for X, where $\sigma'(X) = v$. We now prove ($\Diamond$): $\sigma'(b) \to^*_{E/A} \mathit{true}$. Indeed, $\sigma'(b) = \sigma(b[X/v])$, and the substitution $[X/v]$ is obtained by applying the equation $e \in E$. As E is confluent modulo A and $\sigma(b) \to^*_{E/A} \mathit{true}$, it follows that the equation e is applied at some point in the above reduction. Due again to the confluence, the equations in that reduction can be applied in any order; in particular, e can be applied first, which gives $\sigma(b) \to_{E/A} \sigma'(b) \to^*_{E/A} \mathit{true}$ and proves ($\Diamond$).

Hence $t_1 \to_{R/A} t'_1 \triangleq t[\sigma'(r)]_{p_\alpha}$. Since $\sigma(r) \to^*_{E/A} \sigma'(r)$ ($\sigma(Y) \to^*_{E/A} \sigma'(Y)$ for each variable Y), it follows that $t_2 = t[\sigma(r)]_{p_\alpha} \to^*_{E/A} t[\sigma'(r)]_{p_\alpha} \triangleq u$. The coherence property is now proved.

There remains to prove the faithfulness of the encoding (of language definition into rewrite theory), i.e., $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff $\gamma \xrightarrow{\alpha}_{\mathcal{R}(\mathcal{L})} \gamma'$. By Defs. 1 and 2, $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff $\alpha \triangleq (l \wedge b \Rightarrow r) \in \mathcal{S}$ and $\rho : \mathit{Var} \to \mathcal{T}$ are such that $\gamma =_{\mathcal{T}} \rho(l), \rho(b) =_{\mathcal{T}} \mathit{true}$ and $\gamma' =_{\mathcal{T}} \rho(r)$ (where we empasise that the equality $=_{\mathcal{T}}$ is that of $\mathcal{T}$).

Since the elements in $\mathcal{T}$ are ground $(\Sigma \setminus \Sigma^{\mathsf{Data}}) \cup \mathcal{D}$-terms, their only possible data-subterms are elements in $\mathcal{D}$. It follows that they are E/A-irreducible and hence $\mathcal{T}$ coincides with the algebra of the E/A-canonical forms, i.e. $\mathcal{T} = \mathit{Can}_{\Sigma/(E/A)}$. Hence, each valuation $\rho : \mathit{Var} \to \mathcal{T}$ defines a ground substitution $\sigma_\rho : \mathit{Var} \to T_\Sigma$ such that $\sigma_\rho(X) = \rho(X)$ for each $X \in \mathit{Var}$, and $\mathit{can}_{E/A}(\sigma_\rho(t)) =_A \rho(t)$ for each term $t \in \mathcal{T}_\Sigma(\mathit{Var})$. For instance, if $\rho(I_1) = 3, \rho(I_2) = 5$ then $\rho(I_1 +_{\mathit{Int}} I_2) = \mathcal{T}_{\_+_{\mathit{Int}}\_}(\rho(I_1), \rho(I_2)) = \mathcal{D}_{\_+_{\mathit{Int}}\_}(3, 5) = 8$ and $\sigma_\rho(I_1 +_{\mathit{Int}} I_2) = \_+_{\mathit{Int}}\_(\sigma_\rho(I_1), \sigma_\rho(I_2)) = 3 +_{\mathit{Int}} 5$. The term $3 +_{\mathit{Int}} 5$ is reduced to 8 using the equations E. Conversely, each ground substitution $\sigma : \mathit{Var} \to \mathcal{T}_\Sigma$ defines a valuation $\rho_\sigma : \mathit{Var} \to \mathcal{T}$ such that $\rho_\sigma(X) =_A \mathit{can}_{E/A}(\sigma(X))$ for each $X \in \mathit{Var}$. Thus, if a rule $l \wedge b \Rightarrow r \in \mathcal{S}$ is applicable on a concrete configuration $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ via the valuation ρ and produces γ', then $l \to r \;\mathbf{if}\; b$ in $\mathcal{R}(\mathcal{L})$ is applicable to γ via σ_ρ and produce the same result γ'. Conversely, if a rule $l \to r \;\mathbf{if}\; b$ in $\mathcal{R}(\mathcal{L})$ is applicable on $\gamma \in \mathcal{T}_{\mathit{Cfg}}$ via the substitution σ and produces γ', then $l \wedge b \Rightarrow r \in \mathcal{S}$ is applicable to γ via ρ_σ and produce the same result γ'. Our the faithfulness of the encoding is now proved, and the theorem is proved as well. □

Lemma 1. *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and suppose that $\mathcal{R}(\mathcal{L}) = (\Sigma, E \cup A, 1R \cup 2R)$, where the set of rules is partitioned into two subsets 1R and 2R with $E \cup 1R$ terminating modulo A. If $\gamma \to^*_{(E \cup 1R)/A} \gamma'$ then $\gamma \to^!_{\mathcal{R}(\mathcal{L})} \mathit{can}_{E/A}(\gamma')$, for all $\gamma, \gamma' \in \mathcal{T}_{\mathit{Cfg}}$.*

Proof. By induction on the length i of the derivation $\gamma \to^*_{(E \cup 1R)/A} \gamma' \triangleq \gamma_i$. When $i = 0$, then $\gamma' = \gamma$. Since all data subterms of γ are constants, it follows that γ is irreducible and we obviously have $\gamma \to^!_{\mathcal{R}(\mathcal{L})} \mathit{can}_{E/A}(\gamma) = \gamma$. We assume that $\gamma \to^{*}_{(E\cup \mathit{1R})/A} \gamma_i$ implies $\gamma \to^{!}_{\mathcal{R}(\mathcal{L})} can_{E/A}(\gamma_i)$ (♣), and we want to prove that $\gamma \to^{*}_{(E\cup \mathit{1R})/A} \gamma_i \to_{(E\cup \mathit{1R})/A} \gamma_{i+1}$ implies $\gamma \to^{!}_{\mathcal{R}(\mathcal{L})} can_{E/A}(\gamma_{i+1})$, $i \geq 0$. There are two possibilities for γ_{i+1} to be reached from γ_i:

- $\gamma_i \to_{E/A} \gamma_{i+1}$, i.e γ_{i+1} is reached from γ_i applying an equation. We obtain $\gamma_{i+1} \to^{*}_{E/A} can_{E/A}(\gamma_{i+1}) = can_{E/A}(\gamma_i)$ by the confluence and termination of E and $\gamma \to^{!}_{\mathcal{R}(\mathcal{L})} can_{E/A}(\gamma_{i+1})$ follows directly from (♣).
- $\gamma_i \to_{\mathit{1R}/A} \gamma_{i+1}$, i.e. γ_{i+1} is reached from γ_i applying a rule in *1R*. Since *1R* is ground coherent with respect to E modulo A, there exists a ground term γ_i' such that $can_{E/A}(\gamma_i) \to_{\mathit{1R}/A} \gamma_i'$ and $can_{E/A}(\gamma_{i+1}) =_A can_{E/A}(\gamma_i')$. So, $can_{E/A}(\gamma_i) \to_{\mathit{1R}/A} \gamma_i' \to^{!}_{E/A} can_{E/A}(\gamma_i') =_A can_{E/A}(\gamma_{i+1})$, which implies $\gamma_i \to_{\mathcal{R}(\mathcal{L})} can_{E/A}(\gamma_{i+1})$. We obtain $\gamma \to^{!}_{\mathcal{R}(\mathcal{L})} can_{E/A}(\gamma_{i+1})$ applying (♣).

Now the proof by induction is finished and hence the conclusion of the lemma holds. □

Theorem 4. *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \varepsilon)$ be a two-layered rewrite theory with $(\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R})$ built as in Definition 3 but where the set of rules is partitioned into two subsets 1R and 2R and $E \cup \mathit{1R}$ is terminating modulo A. If $\gamma \twoheadrightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$ then $\gamma \Rightarrow^{+}_{\mathcal{S}} \gamma'$.*
We say that $\mathfrak{R}(\mathcal{L})$ is an approximate encoding *of $\mathcal{L}$.*

Proof. Assume that $\gamma \twoheadrightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$. We have that $\varepsilon(\gamma) \to_{\mathit{2R}/A} \gamma''$, $can_{E/A}(\gamma'') =_A \gamma'$, and $\gamma \to^{!}_{(E\cup \mathit{1R})/A} \varepsilon(\gamma)$ by Defintion 4. Since $\gamma \to^{!}_{(E\cup \mathit{1R})/A} \varepsilon(\gamma)$, there exists n and $\gamma_1, \gamma_2, \ldots, \gamma_n$ such that $\gamma = \gamma_1 \to_{(E\cup \mathit{1R})/A} \gamma_2 \to_{(E\cup \mathit{1R})/A} \cdots \to_{(E\cup \mathit{1R})/A} \gamma_n = \varepsilon(\gamma)$. We obtain $\gamma \to^{!}_{\mathfrak{R}(\mathcal{L})} can_{E/A}(\varepsilon(\gamma))$ (♢) by applying Lemma 1 for $\gamma \to^{*}_{(E\cup \mathit{1R})/A} \gamma_n$. Since $\varepsilon(\gamma)$ is $(E \cup \mathit{1R})$-irreducible, we have $can_{E/A}(\varepsilon(\gamma)) = \varepsilon(\gamma)$ and hence (♢) is equivalent to $\gamma \to^{!}_{\mathfrak{R}(\mathcal{L})} \varepsilon(\gamma)$. From $\varepsilon(\gamma) \to_{\mathit{2R}/A} \gamma''$ and $can_{E/A}(\gamma'') =_A \gamma'$ we have that $\varepsilon(\gamma) \to_{\mathfrak{R}(\mathcal{L})} \gamma'$. We obtain $\gamma \Rightarrow^{*}_{\mathcal{S}} \varepsilon(\gamma)$ by applying Theorem 3 to $\gamma \to^{!}_{\mathfrak{R}(\mathcal{L})} \varepsilon(\gamma)$ and $\varepsilon(\gamma) \to_{\mathfrak{R}(\mathcal{L})} \gamma'$, respectively. Hence we conclude that $\gamma \Rightarrow^{+}_{\mathcal{S}} \gamma'$. □

Lemma 2. *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \varepsilon)$ be an approximate encoding of $\mathcal{L}$. If $\gamma \to_{\mathfrak{R}(\mathcal{L})} \gamma'$, $\gamma \models \pi \wedge \phi$ then there is $\pi' \wedge \phi'$ such that $\pi \wedge \phi \to^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi' \wedge \phi'$ and $\gamma' \models \pi' \wedge \phi'$.*

Proof. Assume $\gamma \to_{\mathfrak{R}(\mathcal{L})} \gamma'$. By the proof of Theorem 4 and Theorem 3 we have $\gamma = \gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n = \gamma'$. Moreover, $\gamma_{n-1} = \varepsilon(\gamma)$, $\alpha_1, \ldots, \alpha_{n-1} \in \mathit{1R}$, and $\alpha_n \in \mathit{2R}$. By Theorem 1 there is $\pi \wedge \phi = \pi_0 \wedge \phi_0 \overset{\alpha_1^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge \phi_1 \overset{\alpha_2^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \overset{\alpha_n^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge \phi_n$ such that $\gamma_i \models \pi_i \wedge \phi_i$. We distinguish two cases:

- $\alpha_1^{\mathfrak{s}}, \ldots, \alpha_{n-1}^{\mathfrak{s}} \in \mathit{1R}^{\mathfrak{s}}$. We have $\phi = \phi_0 = \cdots = \phi_{n-1}$ and hence $\pi_{n-1} \wedge \phi$ is $\mathit{1R}^{\mathfrak{s}}$-irreducible modulo A (otherwise γ_{n-1} is not irreducible by Theorem 2). Since $\alpha_1, \ldots, \alpha_{n-1}$ have been chosen as being minimal w.r.t. $\prec$ it follows that $\alpha_1^{\mathfrak{s}}, \ldots, \alpha_{n-1}^{\mathfrak{s}}$ are minimal w.r.t. $\prec^{\mathfrak{s}}$. Therefore, $\pi_{n-1} \wedge \phi = \varepsilon^{\mathfrak{s}}(\pi \wedge \phi)$ and hence $\pi \wedge \phi \to_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n$.

- There exists $i < n$ such that $\alpha_i \in \mathit{2R}^{\mathfrak{s}}$. Let us consider the smallest i with this property. Assume that $\pi_{i-1} \wedge \phi_{i-1}$ is not $\mathit{1R}^{\mathfrak{s}}$-irreducible. Then there is $\pi_{i-1} \wedge \phi_{i-1} \overset{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi^{\alpha} \wedge \phi^{\alpha}$ with $\alpha^{\mathfrak{s}} \in \mathit{1R}^{\mathfrak{s}}$, i.e. $\alpha^{\mathfrak{s}}$ is unconditional. It follows that $\alpha \prec \alpha_i$ that implies that α_i has not been chosen as minimal w.r.t. $\prec$. Therefore, $\pi_{i-1} \wedge \phi_{i-1}$ is $\mathit{1R}^{\mathfrak{s}}$-irreducible and hence $\pi_{i-1} \wedge \phi = \varepsilon^{\mathfrak{s}}(\pi \wedge \phi)$ and $\pi \wedge \phi \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_i \wedge \phi_i$, as in the previous case. We proceed similarly for the next $i < n$ with $\alpha_i \in \mathit{2R}^{\mathfrak{s}}$, and so on.

□

Theorem 5 (Coverage for Approximate Rewrite Theories). *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \varepsilon)$ be an approximate encoding of $\mathcal{L}$. For every concrete execution $\gamma_0 \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_1 \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_n \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots$ there is a symbolic execution $\pi_0 \wedge \phi_0 \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ such that $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ for $i = 0, 1, \ldots$.*

Proof. Assume $\gamma_0 \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_1 \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_n \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots$. Then, for each $\gamma_i \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_{i+1}$, where $i = 0, 1, \ldots$, (using Lemma 2) there exists a transition $\pi_i \wedge \phi_i \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_{i+1} \wedge \phi_{i+1}$ such that $\gamma_i \models \pi_i \wedge \phi_i$ and $\gamma_i \models \pi_{i+1} \wedge \phi_{i+1}$. □

Lemma 3. *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \varepsilon)$ be an approximate encoding of $\mathcal{L}$ such that $\mathit{1R}$ includes only unconditional rules. If $\pi \wedge \phi \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi' \wedge \phi'$ and there is γ' such that $\gamma' \models \pi' \wedge \phi'$ then there exists γ such that $\gamma \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$ and $\gamma \models \pi \wedge \phi$.*

Proof. Assume $\pi \wedge \phi \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi' \wedge \phi'$. By the proof of Theorem 4 and Theorem 3 we have $\pi \wedge \phi = \pi_0 \wedge \phi_0 \overset{\alpha_1^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge \phi_1 \overset{\alpha_2^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \overset{\alpha_n^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge \phi_n = \pi' \wedge \phi'$, where $\alpha_1^{\mathfrak{s}}, \alpha_2^{\mathfrak{s}}, \ldots, \alpha_{n-1}^{\mathfrak{s}} \in \mathit{1R}^{\mathfrak{s}}$ and $\alpha_n^{\mathfrak{s}} \in \mathit{2R}^{\mathfrak{s}}$. By Theorem 2 there is $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n = \gamma'$, such that $\gamma_i \models \pi_i \wedge \phi_i$. So, based on the fact that $\mathit{1R}$ includes only unconditional rules and $\alpha_1^{\mathfrak{s}}, \ldots, \alpha_{n-1}^{\mathfrak{s}}$ have been chosen ($\varepsilon^{\mathfrak{s}}$) as being minimal w.r.t. $\prec^{\mathfrak{s}}$ which means that $\alpha_1, \ldots, \alpha_{n-1}$ are minimal too w.r.t. $\prec$, we can choose $\gamma = \gamma_0$, where $\gamma_0 \models \pi \wedge \phi$. □

Theorem 6 (precision for approximate rewrite theories). *Let $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ be a language definition and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \varepsilon)$ be an approximate encoding of $\mathcal{L}$ such that $\mathit{1R}$ includes only unconditional rules (hence $\mathit{1R}^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{1R}\}$). For every symbolic execution $\pi_0 \wedge \phi_0 \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ there is a concrete execution $\gamma_0 \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_1 \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_n \rightarrow_{\mathfrak{R}(\mathcal{L})} \cdots$ such that $\gamma_i \in [\![\pi_i \wedge \phi_i]\!]$ for $i = 0, 1, \ldots$.*

Proof. Assume $\pi_0 \wedge \phi_0 \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$. Since each $\pi_i \wedge \phi_i$ is satisfiable, there is γ_i such that $\gamma_i \models \pi_i \wedge \phi_i$, for $i = 0, 1, \ldots$. Then, applying Lemma 3, we may choose $(\gamma_i \mid i \geq 0)$ such that there is a transition $\gamma_i \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma_{i+1}$, for $i = 0, 1, \ldots$ and $\gamma_i \models \pi_i \wedge \phi_i$, for $i = 0, 1, \ldots$. □

B Running paper examples

In this section we exemplify how the use of the 𝕂 framework in order to execute the examples shown in Figure 4. The rewrite theory $\mathcal{R}$ corresponding to CinK definition is obtained as follows:

```
$ kompile cink
```

When running program `counter` we obtain a single solution, that is:

```
$ krun counter.cink
<T>
  <in>
      ListItem(#buffer ( "" ))
      ListItem(#istream ( 0 ))
  </in>
  <out>
      ListItem(#ostream ( 1 ))
      ListItem(#buffer ( "" ))
  </out>
  <control>
      <k>
          .K
      </k>
      <return>
          3
      </return>
      <fstack>
          .List
      </fstack>
      <sideEffects>
          .Bag
      </sideEffects>
      <env>
          counter |-> #symScalLoc(1)
          dec |-> #symScalLoc(3)
          inc |-> #symScalLoc(2)
          main |-> #symScalLoc(4)
      </env>
  </control>
  <genv>
      counter |-> #symScalLoc(1)
      dec |-> #symScalLoc(3)
      inc |-> #symScalLoc(2)
      main |-> #symScalLoc(4)
  </genv>
  <store>
      #symScalLoc(1) |-> 1
      #symScalLoc(2) |-> lambda ( .Decls @ (return (++ counter) ;) )
      #symScalLoc(3) |-> lambda ( .Decls @ (return (-- counter) ;) )
      #symScalLoc(4) |-> lambda ( .Decls @ (return ((inc ( '.Exps(.KList) ))
         + (dec ( '.Exps(.KList) ))) ;) )
  </store>
</T>
```

By default, the tool evaluates the arguments of `+` operator in an arbitrary, but fixed order. The above solution corresponds to the case when `inc()` is evaluated before `dec()`, so the computation of the return value evolves as follows:

$$
\begin{array}{rl}
\texttt{inc}()+\texttt{dec}() & (\textit{heating}) \\
\texttt{inc}() \curvearrowright \square+\texttt{dec}() & (\textit{function lookup}) \\
++\,\texttt{counter} \curvearrowright \square+\texttt{dec}() & (\textit{heating}) \\
\texttt{counter} \curvearrowright ++\square \curvearrowright \square+\texttt{dec}() & (\textit{lookup}) \\
1 \curvearrowright ++\square \curvearrowright \square+\texttt{dec}() & (\textit{cooling}) \\
++1 \curvearrowright \square+\texttt{dec}() & (\textit{increment}) \\
2 \curvearrowright \square+\texttt{dec}() & (\textit{cooling}) \\
2+\texttt{dec}() & (\textit{heating}) \\
\texttt{dec}() \curvearrowright 2+\square & (\textit{function lookup}) \\
--\,\texttt{counter} \curvearrowright 2+\square & (\textit{heating}) \\
\texttt{counter} \curvearrowright --\square \curvearrowright 2+\square & (\textit{lookup}) \\
2 \curvearrowright --\square \curvearrowright 2+\square & (\textit{cooling}) \\
--2 \curvearrowright 2+\square & (\textit{decrement}) \\
1 \curvearrowright 2+\square & (\textit{cooling}) \\
2+1 & (\textit{plus}) \\
2 +_{Int} 1 & (\textit{evaluate } + \textit{ in the integers domain}) \\
3 &
\end{array}
$$

Note that we abstracted the function application and we omitted the application of the rules for statements. If we want to obtain both behaviours, which capture the non-determinism of `+` operation, then we have to compile the definition using the `-superheat` option:

```
$ kompile cink --superheat plus
$ krun counter.cink --search
Search results:

Solution 1:
<T>
```

```
  <in>
      ListItem(#buffer ( "\n" ))
  </in>
  <out>
      ListItem(#buffer ( "" ))
  </out>
  <control>
      <k>
          .K
      </k>
      <return>
          1
      </return>
      <fstack>
          .List
      </fstack>
      <sideEffects>
          .Bag
      </sideEffects>
      <env>
          counter |-> #symScalLoc(1)
          dec |-> #symScalLoc(3)
          inc |-> #symScalLoc(2)
          main |-> #symScalLoc(4)
      </env>
  </control>
  <genv>
      counter |-> #symScalLoc(1)
      dec |-> #symScalLoc(3)
      inc |-> #symScalLoc(2)
      main |-> #symScalLoc(4)
  </genv>
  <store>
      #symScalLoc(1) |-> 1
      #symScalLoc(2) |-> lambda ( .Decls @ (return (++ counter) ;) )
      #symScalLoc(3) |-> lambda ( .Decls @ (return (-- counter) ;) )
      #symScalLoc(4) |-> lambda ( .Decls @ (return ((inc ( ’.Exps(.KList) ))
         + (dec ( ’.Exps(.KList) ))) ;) )
  </store>
</T>

Solution 2:
<T>
  <in>
      ListItem(#buffer ( "\n" ))
```

```
    </in>
    <out>
        ListItem(#buffer ( "" ))
    </out>
    <control>
        <k>
            .K
        </k>
        <return>
            3
        </return>
        <fstack>
            .List
        </fstack>
        <sideEffects>
            .Bag
        </sideEffects>
        <env>
            counter |-> #symScalLoc(1)
            dec |-> #symScalLoc(3)
            inc |-> #symScalLoc(2)
            main |-> #symScalLoc(4)
        </env>
    </control>
    <genv>
        counter |-> #symScalLoc(1)
        dec |-> #symScalLoc(3)
        inc |-> #symScalLoc(2)
        main |-> #symScalLoc(4)
    </genv>
    <store>
        #symScalLoc(1) |-> 1
        #symScalLoc(2) |-> lambda ( .Decls @ (return (++ counter) ;) )
        #symScalLoc(3) |-> lambda ( .Decls @ (return (-- counter) ;) )
        #symScalLoc(4) |-> lambda ( .Decls @ (return ((inc ( '.Exps(.KList) ))
           + (dec ( '.Exps(.KList) ))) ;) )
    </store>
  </T>
```

B.1 Running programs with symbolic values

Translating a language definition such that it can execute programs which operate with symbolic values is done automatically by the 𝕂 tool:

```
  $ kompile cink --backend symbolic
```

The symbolic values are represented as 𝕂 meta-variables (e.g. `A:Int`). Programs containing loops which depend on symbolic values may not terminate. In this case we have to limit the infinite exploration of the state space using a bound:

```
$ krun log.cink --search --bound 3 -cIN=".List" -cPC=true
Search results:

Solution 1:
<path-condition>
  (((A:Int >Int 0) ==Bool true) andBool (((A divInt 2) >Int 0) ==Bool true))
     andBool ((((A divInt 2) divInt 2) >Int 0) ==Bool false)
</path-condition>
<T>
  <in>
      ListItem(#buffer ( "\n" ))
  </in>
  <out>
      ListItem(#buffer ( "" ))
  </out>
  <control>
      <k>
          .K
      </k>
      <return>
          noVal
      </return>
      <fstack>
          .List
      </fstack>
      <sideEffects>
          .Bag
      </sideEffects>
      <env>
          main |-> #symScalLoc(1)
      </env>
  </control>
  <store>
      #symScalLoc(1) |-> lambda ( .Decls @ ((((((int k) ;) ((int x) ;))
        ((x = A) ;)) ((k = 0) ;)) (while ( (x > 0) ) ({ (((++ k) ;) ((x = (x
         / 2)) ;)) }))) )
      #symScalLoc(2) |-> 2
      #symScalLoc(3) |-> (A divInt 2) divInt 2
  </store>
  <genv>
      main |-> #symScalLoc(1)
  </genv>
</T>

Solution 2:
<path-condition>
  ((A:Int >Int 0) ==Bool true) andBool (((A divInt 2) >Int 0) ==Bool false)
</path-condition>
<T>
  <in>
      ListItem(#buffer ( "\n" ))
  </in>
  <out>
      ListItem(#buffer ( "" ))
  </out>
  <control>
      <k>
          .K
      </k>
      <return>
          noVal
      </return>
      <fstack>
          .List
      </fstack>
      <sideEffects>
          .Bag
      </sideEffects>
      <env>
          main |-> #symScalLoc(1)
      </env>
  </control>
  <store>
      #symScalLoc(1) |-> lambda ( .Decls @ ((((((int k) ;) ((int x) ;))
       ((x = A) ;)) ((k = 0) ;)) (while ( (x > 0) ) ({ (((++ k) ;) ((x = (x
         / 2)) ;)) }))) )
      #symScalLoc(2) |-> 1
      #symScalLoc(3) |-> A divInt 2
  </store>
  <genv>
      main |-> #symScalLoc(1)
  </genv>
</T>

Solution 3:
<path-condition>
```

```
    (A:Int >Int 0) ==Bool false
  </path-condition>
  <T>
    <in>
        ListItem(#buffer ( "\n" ))
    </in>
    <out>
        ListItem(#buffer ( "" ))
    </out>
    <control>
        <k>
            .K
        </k>
        <return>
            noVal
        </return>
        <fstack>
            .List
        </fstack>
        <sideEffects>
            .Bag
        </sideEffects>
        <env>
            main |-> #symScalLoc(1)
        </env>
    </control>
    <store>
        #symScalLoc(1) |-> lambda ( .Decls @ ((((((int k) ;) ((int x) ;))
          ((x = A) ;)) ((k = 0) ;)) (while ( (x > 0) ) ({ (((++ k) ;) ((x = (x
           / 2)) ;)) }))) )
        #symScalLoc(2) |-> 0
        #symScalLoc(3) |-> A
    </store>
    <genv>
        main |-> #symScalLoc(1)
    </genv>
  </T>
```

All the solutions contain a special cell named `path-condition` which holds the formula collected during the symbolic execution. We can easily observe that its size increases while the tool explores more and more execution paths.