# Language Definitions as Rewrite Theories

Vlad Rusu, Dorel Lucanu, Traian-Florin Şerbănuţă, Andrei Arusoaie, Andrei Ştefănescu, Grigore Roşu

# Language Definitions as Rewrite Theories

Vlad Rusu[b], Dorel Lucanu[a], Traian-Florin Şerbănuţă[c], Andrei Arusoaie[a,b], Andrei Ştefănescu[d], Grigore Roşu[d]

[a] *"Alexandru Ioan Cuza" University of Iasi, Romania*
[b] *Inria Lille Nord Europe, France*
[c] *University of Bucharest, Romania*
[d] *University of Illinois at Urbana Champaign*

**Abstract**

$\mathbb{K}$ is a formal framework for defining operational semantics of programming languages. The $\mathbb{K}$-Maude compiler translates $\mathbb{K}$ language definitions to Maude rewrite theories. The compiler enables program execution by using the Maude rewrite engine with the compiled definitions, and program analysis by using various Maude analysis tools. $\mathbb{K}$ supports symbolic execution in Maude by means of an automatic transformation of language definitions. The transformed definition is called the *symbolic extension* of the original definition. In this paper we investigate the theoretical relationship between $\mathbb{K}$ language definitions and their Maude translations, between symbolic extensions of $\mathbb{K}$ definitions and their Maude translations, and how the relationship between $\mathbb{K}$ definitions and their symbolic extensions is reflected on their respective representations in Maude. In particular, the results show how analysis performed with Maude tools can be formally lifted up to the original language definitions.

*Keywords:* operational semantics, rewrite theories, symbolic execution, K Framework, Maude

## 1. Introduction

$\mathbb{K}$ [1] is a formal framework for defining operational semantics of programming languages. The version of $\mathbb{K}$ that we are using in this paper [1] includes

[1] $\mathbb{K}$ version 3.4 is available in the online interface `https://fmse.info.uaic.ro/tools/K-3.4/`. A virtual machine running $\mathbb{K}$ 3.4 can be downloaded from `http://www.kframework.org/imgs/releases/kvm-3.4.zip`. The above links are also accessible from the main page of $\mathbb{K}$ `http://www.kframework.org/`.

options that have Maude [2] as a backend: the $\mathbb{K}$ compiler translates a $\mathbb{K}$ definition into a Maude module, and then, the $\mathbb{K}$ runner uses Maude to execute or analyse programs in the defined language.

The Maude backend of $\mathbb{K}$ has been extended with symbolic execution support [3]. Briefly, a $\mathbb{K}$ language definition is automatically transformed into a *symbolic language definition*. The concrete execution of a program using the symbolic definition is the symbolic execution of the same program using the original language definition. The transformation consists of two steps: (1) incorporating *path conditions* in program configurations, and (2) changing the semantics rules to match on *symbolic configurations* and to automatically update the path conditions. A symbolic execution path is called *feasible* if its path conditions are satisfiable. Two results relating concrete and symbolic program executions are proved in [3]: *coverage*, saying that for each concrete execution there is a feasible symbolic execution along the same program path; and *precision*, saying that for each feasible symbolic execution there is a concrete execution along the same program path. If both coverage and precision hold we say that we have a *symbolic extension* relation between a language and a symbolic language.

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 relationship between $\mathbb{K}$ language definitions (including symbolic ones, obtained by the above-described transformation) and their representations in Maude. We also show how the relationship between a language $\mathcal{L}$ and its symbolic extension $\mathcal{L}^{\mathfrak{s}}$ is reflected on their respective representations in Maude. These results ensure that (symbolic) analysis 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 are graphically depicted in the diagrams in Figure 1, where the arrows have the following meaning:

$\xrightarrow{p}$ transformations preserving the property $p$,

$\rightarrowtail^{p}$ relations preserving the property $p$.

The dashed arrows show the results proved in this paper.

In the faithful encoding, each semantics 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

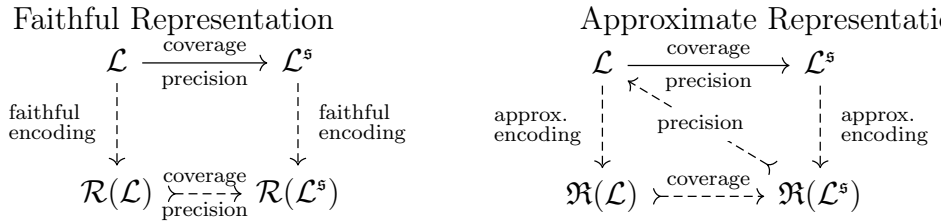


Figure 1: Faithful vs. Approximate representations

to the one generated by the rewrite theory. This ensures that the encoding theories $\mathcal{R}(\mathcal{L})$ and $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ also satisfy the coverage and precision properties relating $\mathcal{L}$ and $\mathcal{L}^{\mathfrak{s}}$. Thus, we can say that the rewrite theory $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ is a *symbolic extension* of $\mathcal{R}(\mathcal{L})$ (in terms of rewrite theories). This means that the symbolic extension and faithful encoding operations commute, as shown by the commuting diagram in the left-hand side of Figure 1.

As a consequence, both positive and negative results of reachability analysis 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 analogous to the results obtained in [4], 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.

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, i.e., $\mathfrak{R}(\mathcal{L})$ is an under-approximation of $\mathcal{L}$. The consequence is that only positive results of reachability analysis on the two-layered rewrite theories can be lifted up to the corresponding language definitions. 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 semantics choices left by the language definition at compile-time. For example, in C and 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. However, this comes at a price. Due to the side effects of some operators, there are C/C++ programs with nondeterministic behaviour. This feature cannot be exhibited with the operand-evaluation rules in the first layer; in order to exhibit the nondeterminism, the rules evaluating the operators must be in the second layer. For programs using operators without side effect, there is no reason to introduce their evaluation rules in the second layer because the result is always the same due the confluence of these rules.

The approximate representations are also useful during the design of the semantics of a language. If one wishes to test the behaviour of some semantical rule, then one can include only that rule in the second layer and use the $\mathbb{K}$ stepper to see the effect of the rule.

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-encoding theories; the precision property holds only in some restricted cases (presented in Theorem 6 later in the paper). Problematic for this are the conditional rules. The symbolic version must execute both branches, when the condition holds and when the condition does not hold. Therefore the rules corresponding to the two cases must be in the second layer, otherwise the first layer could become non-terminating due to iterative statements. This means that some rules which are in the first layer in $\mathfrak{R}(\mathcal{L})$ are in the second layer in $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$. This could affect the order in which the rules are being executed. Recall that the precision property says that for each feasible symbolic execution there is a concrete one taking the same program path. In order to obtain that property we must have appropriate strategies for choosing the execution paths with rules in the first layer. However, the precision property between the approximate symbolic encoding $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ and the language definition $\mathcal{L}$ always holds (Corollary 3). Hence, one can trace symbolic reachability analysis (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.

The present paper extends the results published earlier in [5] in the following directions: the notions of symbolic extensions for language definitions and for rewrite theories are introduced, which allow us to state the results relating language definitions, symbolic execution, and their rewrite theories in a more compact manner. Most importantly, we have increased the expressiveness of language definitions, their symbolic extensions, and their corresponding rewrite theories by allowing axioms such as associativity, commutativity, identity, and combinations thereof for certain operations. This is essential for capturing realistic $\mathbb{K}$ language definitions and their encodings in Maude, since $\mathbb{K}$ definitions use axioms extensively.

*Organisation.* In Section 2 we present our working examples: a partial description of the $\mathbb{K}$ definition for CinK [6] (a kernel of C++) together with two CinK programs.

In Section 3 we present background notions used in the rest of the paper: rewrite theories (the target of our translation of language definitions), and several logics used in the paper, chief among which is Reachability Logic [7], a logic for defining the semantics of programming languages.

In Section 4 we present a formal notion of language definitions, in which the syntax is presented in an algebraic manner and the operational semantics is described using Reachability-Logic rules. This allows us to make our approach independent of the $\mathbb{K}$ language-definition framework and to abstract away particular implementation details of $\mathbb{K}$. We also present the notion of a so-called *symbolic extension* $\mathcal{L}^{\mathbf{s}}$ of a given language $\mathcal{L}$: it is a language whose configurations may contain symbolic values possibly constrained by a *path condition* and whose executions are related to those of $\mathcal{L}$ by so-called *coverage* and *precision* relations.

In Section 5 we present the notion of symbolic extension of a rewrite theory, by analogy with the corresponding notion for language definitions introduced in the previous section.

Section 6 presents the faithful representations of language definitions as rewrite theories. We show that the faithful representation is an *executable* rewrite theory, which satisfies the results graphically depicted in the left-hand side of the diagram in Figure 1: namely, that the faithful representation of a symbolic extension of a language coincides with the symbolic extension of the faithful encoding of the language in question. We also relate our approach

with *rewrite theories modulo SMT* [4], a recently introduced extension of rewrite theories that, under certain conditions, allows one to symbolically execute rewrite theories directly in Maude.

In Section 7 we present the approximate encodings of language definitions as two-layered rewrite theories, and establish the results connecting language definitions, their symbolic versions, and their approximate encodings graphically depicted in the right-hand side of Figure 1.

Section 8 presents the applications of these representations to the compilation of $\mathbb{K}$ language definitions as Maude modules.

Finally, Section 9 presents conclusions and related work.

The proofs of the main theorems together with their auxiliary definitions and lemmas are included in the appendix.

*Acknowledgments.* This work was partially supported by Inria via a CPER grant, by the strategic grant POSDRU/159/1.5/S/137750 "Project Doctoral and Postdoctoral programs support for increased competitiveness in Exact Sciences research" co-financed by the European Social Fund within the Sectorial Operational Program Human Resources Development 2007-2013, by the (Romanian) grant SMIS-CSNR 602-12516 (DAK), and by the NSF grant CCF-1218605.

## 2. Running Example

Our running example is CinK [6], a kernel of the C++ programming language. The $\mathbb{K}$ definition of CinK used in this paper can be found in the $\mathbb{K}$ Framework Github repository: `https://github.com/kframework/cink-semantics/releases/tag/v1.0`. 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 2.

Some of the grammar productions are annotated with $\mathbb{K}$ specific attributes. For example, one feature of C++ expressions is given by the "sequenced before" relation [8], which defines a partial order over the evaluation of subexpressions. This is expressed in $\mathbb{K}$ using the *strict* attribute to specify a nondeterministic evaluation 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.

| | | | |
|---|---|---|---|
| *Exp* | ::= | *Id* \| *Int* | |
| | \| | `++` *Exp* | [*strict*, *prefinc*] |
| | \| | `--` *Exp* | [*strict*, *predec*] |
| | \| | *Exp* `/` *Exp* | [*strict*(*all*(*context*(*rvalue*))), *divide*] |
| | \| | *Exp* `+` *Exp* | [*strict*(*all*(*context*(*rvalue*))), *plus*] |
| | \| | *Exp* `>` *Exp* | [*strict*(*all*(*context*(*rvalue*)))] |
| | \| | *Exp* `=` *Exp* | [*strict*(*1*(*context*(*lvalue*)), 2(*context*(*rvalue*)))] |
| *Stmt* | ::= | *Exp* `;` | [*strict*] |
| | \| | {*Stmts*} | |
| | \| | `while` (*Exp*)*Stmt* | |
| | \| | `return` *Exp* `;` | [*strict*(*all*(*context*(*rvalue*)))] |
| | \| | `if` (*Exp*)*Stmt* `else` *Stmt* | [*strict*(*1*(*context*(*rvalue*)))] |

Figure 2: CinK syntax

Another language feature expressed by $\mathbb{K}$ attributes is given by the classification of expressions into *rvalues* and *lvalues*. For instance, in the expression `x = x + 2` the first occurrence of `x` is an *lvalue* whereas the second one is an *rvalue*. 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* which states that every subexpression must be evaluated as an *rvalue*. Other attributes (*funcall*, ...) are names associated to syntactic productions, which can be used to refer 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, I/O streams, values for program variables, and other additional information. The configuration of CinK (Figure 3) includes the $\langle\rangle_{\mathsf{k}}$ cell containing the code that remains to be executed, which is represented as an associative list of computation tasks $C_1 \curvearrowright C_2 \curvearrowright \ldots$ 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 $\langle\rangle_{\mathsf{stack}}$ and another one $\langle\rangle_{\mathsf{return}}$ for the return values of functions.

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.

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

Figure 3: CinK configuration

| | |
|---|---|
| $I_1$:*Int* `+` $I_2$:*Int* ⇒ $I_1 +_{Int} I_2$ | [*plus*] |
| $I_1$:*Int* `/` $I_2$:*Int* ⇒ $I_1 /_{Int} I_2$ **requires** $I_2 \neq_{Int} 0$ | [*division*] |
| `if(` *true* `)` *St*:*Stmt* `else _` ⇒ *St* | [*if-true*] |
| `if(` *false* `) _ else` *St*:*Stmt* ⇒ *St* | [*if-false*] |
| `while(` *B*:*Exp* `)` *St*:*Stmt* ⇒ `if(` *B* `){` *St* `while(` *B* `)` *St* `else {}}` | [*while*] |
| *V*:*Val* `;` ⇒ · | [*instr-expr*] |
| `{` *Sts*:*Stmts* `}` ⇒ *Sts* | [*block*] |
| $\langle X$:*Id* ⇒ `lval(` *L* `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots$ *X* ↦ *L*:*Loc* $\cdots\rangle_{\mathsf{env}}$ | [*varname*] |
| $\langle$`++lval(` *L*:*Loc* `)` ⇒ `lval(` *L* `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots$ *L* ↦ (*V*:*Int* ⇒ *V* $+_{Int}$ 1) $\cdots\rangle_{\mathsf{store}}$ | [*inc*, *memw*] |
| $\langle$`--lval(` *L*:*Loc* `)` ⇒ `lval(` *L* `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots$ *L* ↦ (*V*:*Int* ⇒ *V* $-_{Int}$ 1) $\cdots\rangle_{\mathsf{store}}$ | [*dec*, *memw*] |
| $\langle$`lval(` *L*:*Loc* `)=` *V*:*Val* ⇒ `lval(` *L* `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots$ *L* ↦ `_` ⇒ *V* $\cdots\rangle_{\mathsf{store}}$ | [*update*, *memw*] |
| `rvalue(lval(` *L*:*Loc* `))` ⇒ `$lookup(` *L* `)` | [*lvalue2rvalue*] |
| $\langle$`$lookup(` *L*:*Loc* `)` ⇒ *V* $\cdots\rangle_{\mathsf{k}}\langle\cdots$ *L* ↦ *V*:*Val* $\cdots\rangle_{\mathsf{store}}$ | [*lookup*, *memr*] |

Figure 4: Subset of rules from the K semantics of CinK

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 performs an unrolling 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*.

*The* $\mathbb{K}$ *Notation.* A $\mathbb{K}$ *rule* is a topmost rewrite rule specifying transitions between configurations. Typically, only a small part of the configuration is changed by a rule, while a larger context is needed to check the preconditions of the rule. The $\mathbb{K}$ notation simplifies the writing of such rules through *configuration abstraction* and *local rewriting*. *Configuration abstraction* allows one to only specify the parts transformed by the rule using static positional information about the cells (specified by the initial configuration) to fill in the missing structure. For instance, the (abstract) rule for addition, shown in Figure 4, represents the (concrete) rule

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

where $+_{Int}$ is the mathematical operation for addition.

*Local rewriting* addresses the fact that although a larger context is needed to make sure a rule applies and to collect data about the application, usually only minor changes occur. To allow a more compact, precise, and less error-prone specification of transitions, $\mathbb{K}$ lets the user specify precisely where the changes in the matched configuration take place by allowing the rewrite symbol $\Rightarrow$ to occur inside a matching context.

Consider, for example, the *lookup* rule in Figure 4. If the next execution task consists in looking up for the value of location $L$ in the store, and the location $L$ is mapped to a value $V$ in the store, then the rule locally (i.e., at the top of the $k$ cell) replaces the current execution task by the computed value $V$, without otherwise modifying the matching context.

$$\langle \texttt{\$lookup}\ (L{:}Loc) \Rightarrow V\ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ L \mapsto V{:}Val\ \cdots\rangle_{\mathsf{store}} \quad [lookup, memr]$$

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 *desugared* topmost rewrite rule corresponding to the *lookup* rule is automatically obtained by replicating the context and filling in the missing variables and cells:

$$\begin{array}{rr} & \langle\langle \texttt{\$lookup}\ (L{:}Loc) \curvearrowright K\rangle_{\mathsf{k}}\ \langle L \mapsto V{:}Val\ S\rangle_{\mathsf{store}}\ C\rangle_{\mathsf{cfg}} \\ \Rightarrow & \langle\langle V \curvearrowright K\rangle_{\mathsf{k}}\ \langle L \mapsto V \qquad S\rangle_{\mathsf{store}}\ C\rangle_{\mathsf{cfg}} \end{array}$$

To address the fact that a rule might induce changes in several places in the configuration, $\mathbb{K}$ allows the rewrite symbol to occur multiple times in the matching context (as long as it is not nested). This can be seen in several of the rules in Figure 4.

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

$$A_1 \texttt{ / } A_2 \Rightarrow \texttt{rvalue}\ (A_1) \curvearrowright \square \texttt{ / } A_2 \tag{1}$$
$$A_1 \texttt{ / } A_2 \Rightarrow \texttt{rvalue}\ (A_2) \curvearrowright A_1 \texttt{ / } \square \tag{2}$$
$$\texttt{rvalue}\ (I_1) \curvearrowright \square \texttt{ / } A_2 \Rightarrow I_1 \texttt{ / } A_2 \tag{3}$$
$$\texttt{rvalue}\ (I_2) \curvearrowright A_1 \texttt{ / } \square \Rightarrow A_1 \texttt{ / } I_2 \tag{4}$$

```
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;
  cin >> x;
  k = 0;
  while (x > 0) {
    ++k;
    x = x / 2;
  }
}
```

b) The program `log`

Figure 5: Two C++ programs

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

We shall be using the programs in Figure 5 in the sequel. The program `counter` is nondeterministic; the nondeterminism arises from the unspecified evaluation order for the arguments of the `+` operation and from the side effects in its arguments. The program `log` takes as input an integer value $A$ and computes in the variable `k` the value $\lfloor \log_2(A) \rfloor$, where $\lfloor \_ \rfloor$ denotes the integer part of a real number. In Section 8 we show how the behaviours of these programs can be analysed using our tool.

## 3. Background

In this section we present background material on rewrite theories (Section 3.1) and on the logics used in the paper, including (topmost) matching logic and reachability logic (Section 3.2.2).

### *3.1. Rewrite Theories*

We consider an extended notion of rewrite theories, namely, *rewrite theories with entailment*, which allow us to capture both standard rewrite theories and rewrite theories modulo a builtin subtheory [4] that we use for encoding language definitions.

A **rewrite theory with entailment** $\mathcal{R} = (\Sigma, E \cup A, R, \vdash)$ consists of a signature $\Sigma$, a set of equations $E$, a set of axioms $A$, e.g., associativity, commutativity, identity or combinations of these, 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 *condition*,

and an entailment relation $\vdash$ which is sound w.r.t. the semantic interpretation of the formulas in the standard model, i.e., $E \cup A \vdash b$ implies $T_{E \cup A} \models b$. We often write $\vdash b$ for $E \cup A \vdash b$. We assume there is a distinguished sort *Bool* such that each term $b \in T_{\Sigma,Bool}$ is a particular case of formula. For the rewrite theories supported by the Maude language [2], the formulas used in conditions are of the form $(\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_k w_k \rightarrow w_k')$, the three conjuncts being respectively equations, memberships, and rewrite conditions. Here we let more freedom in choosing the condition formulas, e.g., the conditions could be first-order formulas and the entailment $\vdash$ is checked by an external prover.

The relation $\rightarrow_{R/A}$ denotes the one-step (ground) rewriting relation defined by applying a rule from $R$ modulo axioms $A$ over ground terms: $u \rightarrow_{R/A} v$ iff there are terms $u', v'$, a rule $l \rightarrow r$ **if** $b$ in $R$, a position $p$ in $u'$, and a (ground) substitution $\sigma$ such that $u =_A u'$, $v =_A v'$, $u'|_p = \sigma(l)$, $v' = u[\sigma(r)]_p$, and $\vdash \sigma(b)$. We use $t|_p$ to denote the subterm of $t$ at position $p$, $t[u]_p$ to denote the term obtained from $t$ by replacing the subterm at position $p$ with $u$, and $var(t)$ to denote the set of variables occurring in $t$.

*Remark* 1. If conditions $b$ are of the form $\bigwedge_i u_i \rightarrow v_i$ and we define $\vdash$ by $\vdash \bigwedge_i u_i \rightarrow v_i$ iff $\vdash_n \bigwedge_i u_i \rightarrow v_i$ for some $n \geq 0$, where $\vdash_n$ and $\rightarrow_{R/A,n}$ are inductively defined as follows:

- $\rightarrow_{R/A,0} = \emptyset$, $\vdash_0 = \emptyset$;
- $\rightarrow_{R/A,n+1} = \rightarrow_{R/A,n} \cup \{(u, v) \mid (\exists p)(\exists l \rightarrow r \textbf{ if } b \in R)$
  $u|_p =_A \sigma(l), v =_A u[\sigma(r)]_p, \vdash_n \sigma(b)\}$
  $\vdash_{n+1} = \vdash_n \cup \{\bigwedge_i u_i \rightarrow v_i \mid u_i \rightarrow^*_{R/A,n} v_i \text{ for all } i\}$

then $\rightarrow_{R/A}$ is the standard inductive definition for the one-step rewriting relation in rewriting logic (see, e.g., [9]).

We are only interested in rewrite theories $\mathcal{R}$ that are **executable**, i.e.,

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 $can_{E/A}(t)$ that is unique modulo the axioms $A$;
3. $R$ is **ground coherent w.r.t.** $E$ **modulo** $A$ [10]: for all $t, t_1 \in T_\Sigma$ with $t \rightarrow_{R/A} t_1$ there is $t_2 \in T_\Sigma$ s.t. $can_{E/A}(t) \rightarrow_{R/A} t_2$ and $can_{E/A}(t_1) =_A can_{E/A}(t_2)$;

4. $\vdash$ is decidable.

We note that the last condition is specific to rewrite theories with entailment. The **rewriting relation** $\rightarrow_{\mathcal{R}}$ defined by an executable rewrite theory $\mathcal{R}$ is: $t_1 \rightarrow_{\mathcal{R}} t_2$ iff $can_{E/A}(t_1) \rightarrow_{R/A} t'_2$ and $can_{E/A}(t'_2) = t_2$. This is equivalent to $\rightarrow_{R/(E \cup A)}$ due to confluence, termination and coherence. We write $t_1 \xrightarrow{\alpha}_{\mathcal{R}} t_2$ to emphasise that $\alpha \triangleq (l \rightarrow r \textbf{ if } b) \in R$ is applied in the rewriting step $can_{E/A}(t_1) \rightarrow_{R/A} t'_2$. The executability conditions on $\mathcal{R}$ ensure that the set of successors of a term by the rewriting relation $\rightarrow_{\mathcal{R}}$ is computable.

The set of $E/A$-canonical forms can be organised as a $\Sigma$-algebra $Can_{\Sigma,E/A}$, where a functional symbol $f \in \Sigma_{s_1 \ldots s_n, s}$ is interpreted as the function that sends a tuple $(t_1, \ldots, t_n)$ of $E/A$-canonical terms of appropriate sorts to the the $E/A$-canonical form of $f(t_1, \ldots, t_n)$. $Can_{\Sigma,E/A}$ is isomorphic to the initial algebra $T_{\Sigma,E \cup A}$ (see, e.g., [11]).

### *3.2. Logics*

In this section we introduce the logics employed in the paper.

#### *3.2.1. First-Order Logic*

We assume the reader is familiar with algebraic specifications and the basics of (many-sorted) First-Order Logic (FOL). We denote a many-sorted first-order signature by a pair $(\Sigma, \Pi)$, where $\Sigma$ is an algebraic signature and $\Pi$ is a set of predicates. The set of FOL formulas over $(\Sigma, \Pi)$ is defined as:

$$\phi ::= \mathit{True} \mid p(t_1, \ldots, t_n) \mid \neg\phi \mid \phi \wedge \phi \mid (\exists V)\phi$$

where $p$ is a predicate symbol in $\Pi$, each $t_i$ is a term in $T_\Sigma(\mathit{Var})$ of appropriate sort, and $V$ is a subset of $\mathit{Var}$. A $(\Sigma, \Pi)$-model consists of a $\Sigma$-algebra $M$ together with a set $M_p$ for each predicate $p \in \Pi$. A valuation $\rho$ maps variables from $\mathit{Var}$ to elements in $M$. A substitution $\sigma$ is a particular valuation, where $M$ is $T_\Sigma(\mathit{Var})$. We let $dom(\sigma)$ denote the domain of $\sigma$, $\{x \in \mathit{Var} \mid \sigma(x) \neq x\}$, and $ran(\sigma)$ denote the range of $\sigma$, $\{\sigma(x) \mid x \in dom(\sigma)\}$.

Next, we give a construction of a free extension of a FOL model with respect to a subsignature, which will be used later in the paper.

**Definition 1.** Let $(\Sigma, \Pi)$ be an $S$-sorted FOL signature and $M, M'$ be two $(\Sigma, \Pi)$-models. A $(\Sigma, \Pi)$**-model-morphism** $h : M \rightarrow M'$ is an $S$-sorted family of functions $(h_s : M_s \rightarrow M'_s \mid s \in S)$ such that

- $h_s(M_f(a_1, \ldots, a_n)) = M'_f(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ for each function symbol $f : s_1 \times \ldots \times s_n \to s$ in $\Sigma$ and $(a_1, \ldots, a_n) \in M_{s_1} \times \ldots \times M_{s_n}$, and

- $(a_1, \ldots, a_n) \in M_p$ implies $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in M'_p$ for each predicate symbol $p : s_1 \times \ldots \times s_n$ in $\Pi$ and $(a_1, \ldots, a_n) \in M_{s_1} \times \ldots \times M_{s_n}$.

*Example* 1. Let us consider a signature $(\Sigma, \Pi)$ which includes a sort *Int*, and two $(\Sigma, \Pi)$ models $M$ and $M'$ which include the carrier sets $\mathbb{Z}$ and $\mathbb{Z}_{32}$ (32-bit integers), respectively. Then, a $(\Sigma, \Pi)$-model-morphism $h : M \to M'$ includes a function $h_{Int} : \mathbb{Z} \to \mathbb{Z}_{32}$ which sends an integer $x$ into a 32-bit representation. The function $h_{Int}$ must satisfy $h_{Int}(M_+(x, y)) = M'_+(h_{Int}(x), h_{Int}(y))$.

In this paper we use $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$ to distinguish the (sub)signature of **data** sorts in a programming language from the other **non-data** constructs of the language.

**Definition 2.** Let $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d}) \subset (\Sigma, \Pi)$ be a signature inclusion and $M$ be a $(\Sigma, \Pi)$-model. Then $M{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})}$ is the $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$-model $M^\mathfrak{d}$ defined as follows:

- $M^\mathfrak{d}_s = M_s$ for each $\Sigma^\mathfrak{d}$-sort;

- $M^\mathfrak{d}_f = M_f$ for each function symbol $f$ in $\Sigma^\mathfrak{d}$;

- $M^\mathfrak{d}_p = M_p$ for each predicate symbol $p$ in $\Pi^\mathfrak{d}$.

*Example* 2. Let $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$ be a signature which includes some data sorts (like *Int*, *Bool*, *String*, ...) and is included ($\subset$) in a signature $(\Sigma, \Pi)$. If $M$ is a $(\Sigma, \Pi)$-model then $M{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})}$ is a $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$-model. This model interprets data sorts and their operations in the same way these are interpreted in $M$.

**Definition 3.** Let $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d}) \subset (\Sigma, \Pi)$ be a signature inclusion and $h : M \to M'$ be a $(\Sigma, \Pi)$-model-morphism. Then $h{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})}$ is the $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$-model-morphism $h' : M{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})} \to M'{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})}$ defined by $h'_s = h_s$ for each $\Sigma^\mathfrak{d}$-sort $s$.

**Definition 4.** Let $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d}) \subset (\Sigma, \Pi)$ be a signature inclusion and $\mathcal{D}$ be a $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$-model. A $(\Sigma, \Pi)$-model $M$ is **free over** $\mathcal{D}$ (in the category of $(\Sigma, \Pi)$-models) if $M{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})} = \mathcal{D}$ and for any $(\Sigma, \Pi)$-model $M'$ and $(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})$-model-morphism $h : \mathcal{D} \to M'{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})}$ there is a unique $(\Sigma, \Pi)$-model-morphism $h^\# : M \to M'$ such that $h^\#{\restriction}_{(\Sigma^\mathfrak{d}, \Pi^\mathfrak{d})} = h$.

Since the free model $M$ over $\mathcal{D}$ is unique up to an isomorphism, we often denote it by $\mathcal{D}\!\upharpoonright^{(\Sigma,\Pi)}$. A constructive definition for $\mathcal{D}\!\upharpoonright^{(\Sigma,\Pi)}$ is given by the following result, whose proof is standard.

**Proposition 1.** *Let $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}}) \subset (\Sigma, \Pi)$ be a signature inclusion and $\mathcal{D}$ be a $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$-model. We assume that the result sort for any function symbol $f$ in $\Sigma \setminus \Sigma^{\mathfrak{d}}$ is not a $\Sigma^{\mathfrak{d}}$-sort. Then the model $M$, defined as follows:*

- *for each item (sort, function/predicate symbol) $o \in (\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$, $M_o = \mathcal{D}_o$;*
- *for each sort $s$ in $\Sigma \setminus \Sigma^{\mathfrak{d}}$, $M_s$ is the set of ground $(\Sigma \setminus \Sigma^{\mathfrak{d}})(\mathcal{D})$-terms;*
- *for each function symbol $f$ in $\Sigma \setminus \Sigma^{\mathfrak{d}}$, $M_f$ is the term constructor $f$ such that for all $(t_1, \ldots, t_n)$, $M_f(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ (recall that the result sort of $f$ does not belong to $\Sigma^{\mathfrak{d}}$ by the hypotheses);*
- *for each predicate symbol $p$ in $\Pi \setminus \Pi^{\mathfrak{d}}$, $M_p$ is the empty set,*

*is free over $\mathcal{D}$.*

*Example* 3. Let $\mathcal{D}$ be the data domain over which the semantics of CinK is defined. For instance, $\mathcal{D}$ interprets sorts like *Bool* and *Int*, interprets operations like $+_{Int} \in \Sigma^{\mathfrak{d}}$ and predicates $<_{Int} \in \Pi^{\mathfrak{d}}$. Then $\mathcal{D}\!\upharpoonright^{(\Sigma,\Pi)}$ includes the CinK code and the configurations obtained during the execution of programs. Note that in general $\mathcal{D}\!\upharpoonright^{(\Sigma,\Pi)}$ is not necessarily the term algebra built over $\mathcal{D}$ because it could be required that some non-data operations must satisfy axioms like associativity (e.g., $\curvearrowright$).

*3.2.2. Matching Logic and Reachability Logic*

Our generic notion of language-definition framework, introduced below in Section 4, uses (topmost) *Matching Logic (ML)* [12] for the static parts of language definitions, and *Reachability Logic (RL)* [13] for their dynamic parts.

**Definition 5** (ML signature)**.** An ML signature $\Lambda = (\Sigma, \Pi, \mathit{Cfg})$ is a first-order signature $(\Sigma, \Pi)$ together with a distinguished sort *Cfg* for **configurations**.

**Definition 6** (ML formula)**.** Given an ML signature $\Lambda = (\Sigma, \Pi, \mathit{Cfg})$, the set of **ML formulas** over $\Lambda$, also called **patterns**, is defined by

$$\varphi ::= \pi \mid \mathit{True} \mid p(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid (\exists X)\varphi$$

where $\pi$ ranges over $T_{\Sigma,Cfg}(Var)$, $p$ ranges over predicate symbols $\Pi$, each $t_i$ ranges over $T_\Sigma(Var)$ of appropriate sorts, and $X$ over finite subsets of $Var$. An **elementary pattern** is an ML formula of the form $\pi \wedge \phi$ and a **basic pattern** is an ML formula $\pi \in T_{\Sigma,Cfg}(Var)$.

We sometimes say "ML formula" instead of "ML formula over the ML signature $\Lambda$" when the signature $\Lambda$ is clear from the context.

*Example* 4. Recall the semantical rules defining CinK from Figure 4. Each rule, after desugaring, is made up from two components separated by the $\Rightarrow$ symbol. Those components are CinK configurations, possibly accompanied by a logical constraint. For instance, the rule for division (after being completed with cells by the configuration desugaring mechanism) is:

$$\langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$$
$$\Rightarrow$$
$$\langle\langle I_1 /_{Int} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}} \textbf{ requires } I_2 \neq_{Int} 0$$

The left-hand side of the rule, $\langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$, is a term of sort $T_{\Sigma,Cfg}(Var)$ (cf. Example 7), which is a basic pattern (cf. Definition 6). The right-hand side can also be seen as a basic pattern. The left hand side together with the condition form an elementary pattern:

$$\langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0.$$

We extend the notion of (free) variables $var()$ from terms to ML formulas. For elementary patterns $\varphi \triangleq \pi \wedge \phi$ we have $var(\varphi) = var(\pi) \cup var(\phi)$. For simplicity we write $var(\pi, \phi)$ to denote $var(\pi) \cup var(\phi)$, and for all patterns $\varphi$ and $\varphi'$ we let $var(\varphi, \varphi') \triangleq var(\varphi) \cup var(\varphi')$.

**Definition 7** (ML satisfaction relation)**.** Given an ML signature $\Lambda = (\Sigma, \Pi, Cfg)$, $\mathcal{T}$ a $(\Sigma, \Pi)$-model, $\varphi$ an ML formula, $\gamma \in \mathcal{T}_{Cfg}$ a configuration, and $\rho : Var \to \mathcal{T}$ a valuation, the satisfaction relation $(\gamma, \rho) \models_{\mathrm{ML}} \varphi$ is defined as follows:

1. $(\gamma, \rho) \models_{\mathrm{ML}} \pi$ iff $\rho(\pi) = \gamma$;
2. $(\gamma, \rho) \models_{\mathrm{ML}} True$;
3. $(\gamma, \rho) \models_{\mathrm{ML}} p(t_1, \ldots, t_n)$ iff $(\rho(t_1), \ldots, \rho(t_n)) \in M_p$;
4. $(\gamma, \rho) \models_{\mathrm{ML}} \neg\varphi$ iff $(\gamma, \rho) \models_{\mathrm{ML}} \varphi$ does not hold;
5. $(\gamma, \rho) \models_{\mathrm{ML}} \varphi_1 \wedge \varphi_2$ iff $(\gamma, \rho) \models_{\mathrm{ML}} \varphi_1$ and $(\gamma, \rho) \models_{\mathrm{ML}} \varphi_2$; and
6. $(\gamma, \rho) \models_{\mathrm{ML}} (\exists X)\varphi$ iff there is $\rho' : Var \to \mathcal{T}$ with $\rho'(y) = \rho(y)$, for all $y \notin X$, such that $(\gamma, \rho') \models_{\mathrm{ML}} \varphi$.

The $\mathcal{T}$**-semantics** of a formula is given by $[\![\varphi]\!]_{\mathcal{T}} = \{\gamma \in \mathcal{T}_{Cfg} \mid \exists \rho : Var \to \mathcal{T}, (\gamma, \rho) \models \varphi\}$. A ML formula $\varphi$ is **valid**, denoted by $\models_{\mathrm{ML}} \varphi$, if for all $(\gamma, \rho)$ it holds that $(\gamma, \rho) \models_{\mathrm{ML}} \varphi$.

We often write only $[\![\varphi]\!]$ if the model $\mathcal{T}$ is understood from the context.

*Example* 5. Consider the ML formula

$$\varphi(\triangleq \pi \wedge \phi) \triangleq \langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}} \wedge I_2 \neq_{Int} 0$$

and a (configuration) term in $\mathcal{T}_{Cfg}$

$$\gamma \triangleq \langle\langle 2 \;/\; 1 \curvearrowright \cdot\rangle_{\mathsf{k}}\langle\cdot\rangle_{\mathsf{env}}\langle\cdot\rangle_{\mathsf{store}}\langle\cdot\rangle_{\mathsf{stack}}\langle 2\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$$

If $\rho : Var \to \mathcal{T}$ is a valuation such that $\rho(I_1) = 2$, $\rho(I_2) = 1$, $\rho(C) = \cdot$, $\rho(E) = \cdot$, $\rho(S) = \cdot$, $\rho(T) = \cdot$, and $\rho(V) = 2$, then $(\gamma, \rho) \models \varphi$ since $\rho(\pi) = \gamma$ and $\rho(I_2) = 2 \neq_{Int} 0$.

The dynamics of programs in a language are given by Reachability Logic (RL) formulas, which are pairs of ML formulas:

**Definition 8** (RL formula)**.** Given an ML signature $\Lambda = (\Sigma, \Pi, Cfg)$, an **RL formula**, a.k.a. **reachability rule**, (over $\Lambda$) is an expression of the form $\varphi \Rightarrow \varphi'$ where $\varphi$ and $\varphi'$ are ML formulas (over $\Lambda$).

The $\mathbb{K}$ rules shown in Figure 4 are examples of RL formulas. For instance, the rule from Example 4 can be written as

$$\langle\langle I_1 \;/\; I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}} \;\wedge\; I_2 \neq_{Int} 0$$
$$\Rightarrow$$
$$\langle\langle I_1 /_{Int} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$$

**Definition 9** (RL transition system)**.** Let $\Lambda = (\Sigma, \Pi, Cfg)$ be an ML signature, $\mathcal{T}$ be a $(\Sigma, \Pi)$-model, and let $\mathcal{S}$ be a set of reachability rules over $\Lambda$. Then $\mathcal{S}$ induces a transition system $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}})$, where $\Rightarrow_{\mathcal{S}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ is defined by $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ iff there is $\alpha \triangleq \varphi \Rightarrow \varphi'$ in $\mathcal{S}$ and $\rho : Var \to \mathcal{T}$ with $(\gamma, \rho) \models_{\mathrm{ML}} \varphi$ and $(\gamma', \rho) \models_{\mathrm{ML}} \varphi'$. We sometimes write $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ in order to explicitly state that the applied rule is $\alpha$.

*Example* 6. Consider

$$\gamma \triangleq \langle\langle \texttt{if}\,(\mathit{true})\;\texttt{skip}\;\texttt{else}\;\texttt{skip}\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 7\rangle_{\mathsf{env}}\langle 7 \mapsto 23\rangle_{\mathsf{store}}\langle\rangle_{\mathsf{stack}}\langle\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$$

and

$$\gamma' \triangleq \langle\langle \texttt{skip}\rangle_{\mathsf{k}}\langle \texttt{x} \mapsto 7\rangle_{\mathsf{env}}\langle 7 \mapsto 23\rangle_{\mathsf{store}}\langle\rangle_{\mathsf{stack}}\langle\rangle_{\mathsf{return}}\rangle_{\mathsf{cfg}}$$

two CinK configurations. Then there is a transition $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$, where $\alpha$ is the rule *if-true* in Figure 4.

## 4. Language Definitions and their Symbolic Extensions

### *4.1. Language Definitions*

**Definition 10.** A **language definition** is a tuple $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ where:

- $(\Sigma, \Pi, \mathit{Cfg})$ is an ML signature
- $\mathcal{T}$ is a model of $(\Sigma, \Pi, \mathit{Cfg})$
- $\mathcal{S}$ is a finite set of RL formulas, of the form $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$, where $\pi_1 \wedge \phi_1, \pi_2 \wedge \phi_2$ are elementary patterns over the signature $(\Sigma, \Pi, \mathit{Cfg})$.

*Example* 7. Recall the syntax and semantics of CinK from Figures 2 and 4. Every non-terminal in the syntax is a sort in $\Sigma$ (e.g. *Exp*). Every syntax production defines a new operation symbol; for instance, the corresponding operation for the production $\mathit{Exp} ::= \mathit{Exp}$ + $\mathit{Exp}$ is $\_+\_ : \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}$, which takes two terms of sort *Exp* and returns a new term of sort *Exp*. The sort *Cfg* corresponds to CinK's configuration which is shown in Figure 3. *Cfg* terms can be constructed using the operation

$$\langle \langle \_ \rangle_{\mathsf{k}} \langle \_ \rangle_{\mathsf{env}} \langle \_ \rangle_{\mathsf{store}} \langle \_ \rangle_{\mathsf{stack}} \langle \_ \rangle_{\mathsf{return}} \rangle_{\mathsf{cfg}} : K \times \mathit{Map} \times \mathit{Map} \times \mathit{List} \times K \to \mathit{Cfg}.$$

Every $\mathbb{K}$ rule shown in Figure 4 is included in $\mathcal{S}$. Note that every $\mathbb{K}$ rule is an RL formula (see Example 4).

We also assume a (strict) subsignature $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ of $(\Sigma, \Pi)$ for the language's data types (integers, lists, etc) and a $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$-model $\mathcal{D}$ such that $\mathcal{T}$ restricted to $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ equals $\mathcal{D}$, i.e., $\mathcal{T}\restriction_{(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})} = \mathcal{D}$. The sort *Cfg* is not a data sort. We sometimes call language definitions *languages* for simplicity. A language definition $\mathcal{L}$ naturally induces a transition system $(\mathcal{T}_{\mathit{Cfg}}, \Rightarrow_{\mathcal{S}})$, where $\Rightarrow_{\mathcal{S}}$ is given in Definition 9.

### *4.2. Symbolic Extension of a Language Definition*

We assume a countably infinite sorted set of variables $\mathit{Var}^{\mathfrak{d}}$ of data sorts.

**Definition 11** (Symbolic extension of language definition)**.** Given a language definition $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$, we say that another language definition $\mathcal{L}^{\mathfrak{s}} = ((\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, \mathit{Cfg}^{\mathfrak{s}}), \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ is a **symbolic extension** of $\mathcal{L}$ if the semantic domain $((\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, \mathit{Cfg}^{\mathfrak{s}}), \mathcal{T}^{\mathfrak{s}})$ satisfies:

1. $\Sigma^{\mathfrak{s}}$ contains two new sorts: $\mathit{Cfg}^{\mathfrak{s}}$ and $\mathit{Bool}$[2].
   The operations of sort $\mathit{Bool}$ include the usual propositional items ($\mathit{true}$, $\mathit{false}, \wedge, \neg$) as well as an operation $p : s_1 \times \cdots \times s_n \to \mathit{Bool}$ for each predicate $p \in \Pi_{s_1 \ldots, s_n}$. The unique operation of sort $\mathit{Cfg}^{\mathfrak{s}}$ is its constructor $\wedge^{\mathfrak{s}} : \mathit{Cfg} \times \mathit{Bool} \to \mathit{Cfg}^{\mathfrak{s}}$. We naturally identify terms of the sort $\mathit{Bool}$ with the corresponding FOL formulas. We extend the notation $[\![\cdot]\!]$ defined on ML formulas, to the corresponding terms of sort $\mathit{Cfg}^{\mathfrak{s}}$, by letting $[\![\pi \wedge^{\mathfrak{s}} \phi]\!] \triangleq [\![\pi \wedge \phi]\!]$.
2. $\Pi^{\mathfrak{s}}$ consists of one predicate $\mathit{sat}$, which takes one argument of sort $\mathit{Bool}$.
3. $\mathcal{T}^{\mathfrak{s}}$ interprets the elements of $\Sigma^{\mathfrak{s}}$ and $\Pi^{\mathfrak{s}}$ as follows:
   - all sorts $s$ in $\Sigma$ are interpreted as sets of terms in $T_{\Sigma(\mathcal{D}),s}(\mathit{Var}^{\mathfrak{d}})$;
   - the sort $\mathit{Bool}$ is interpreted as a set of terms in $T_{\Sigma^{\mathfrak{s}}(\mathcal{D}),\mathit{Bool}}(\mathit{Var}^{\mathfrak{d}})$;
   - the sort $\mathit{Cfg}^{\mathfrak{s}}$ is interpreted as sets of terms of sort $\mathit{Cfg}^{\mathfrak{s}}$, of the form $\pi \wedge^{\mathfrak{s}} \phi$, where $\pi \in T_{\Sigma(\mathcal{D}),\mathit{Cfg}}(\mathit{Var}^{\mathfrak{d}})$ and $\phi \in T_{\Sigma^{\mathfrak{s}}(\mathcal{D}),\mathit{Bool}}(\mathit{Var}^{\mathfrak{d}})$,
   - all the operations in $\Sigma^{\mathfrak{s}}$ are interpreted syntactically;
   - the (unique) predicate $\mathit{sat} \in \Pi^{\mathfrak{s}}_{\mathit{Bool}}$ is interpreted as the theoretical satisfiability predicate for FOL formulas represented as terms of sort $\mathit{Bool}$, i.e., $\phi \in \mathcal{T}^{\mathfrak{s}}_{\mathit{sat}}$ iff the FOL formula represented by $\phi$ is satisfiable;

and the rules $\mathcal{S}^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathcal{S}\}$ are defined such that the map $\alpha \mapsto \alpha^{\mathfrak{s}}$ is a bijection and the following two properties hold:

**Coverage** If $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$ with $\pi \wedge^{\mathfrak{s}} \phi \in \mathcal{T}^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}$, then there exists $\pi' \wedge^{\mathfrak{s}} \phi' \in \mathcal{T}^{\mathfrak{s}}_{\mathit{Cfg}^{\mathfrak{s}}}$ such that $\pi \wedge^{\mathfrak{s}} \phi \overset{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$ and $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$.

**Precision** If $\pi \wedge^{\mathfrak{s}} \phi \overset{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$ and $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$ then there exists a configuration $\gamma$ such that $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$.

*Example* 8. As pointed in Definition 11, the symbolic extension of CinK contains two new sorts $\mathit{Cfg}^{\mathfrak{s}}$ and $\mathit{Bool}$. The first one corresponds to symbolic configurations, while the second one is intended to allow constraints over the symbolic values inside such a symbolic configuration. For instance, $\pi \wedge^{\mathfrak{s}} \phi \triangleq \langle \langle \mathsf{a} \;=\; a \rangle_{\mathsf{k}} \langle \cdot \rangle_{\mathsf{env}} \langle \cdot \rangle_{\mathsf{store}} \langle \cdot \rangle_{\mathsf{stack}} \langle \cdot \rangle_{\mathsf{return}} \rangle_{\mathsf{cfg}} \wedge^{\mathfrak{s}} a > 0$, where $a \in \mathit{Var}^{\mathfrak{d}}$, is a symbolic configuration with $\pi \in T_{\Sigma(\mathcal{D}),\mathit{Cfg}}(\mathit{Var}^{\mathfrak{d}})$ and $\phi \in T_{\Sigma^{\mathfrak{s}}(\mathcal{D}),\mathit{Bool}}(\mathit{Var}^{\mathfrak{d}})$.

[2]Not to be confused with the Boolean datatype that some languages may implement.

The sort *Bool* together with its operations belongs to $\Sigma^{\mathfrak{s}\mathfrak{d}}$, and *sat* is in $\Pi^{\mathfrak{s}\mathfrak{d}}$, i.e., they belong to the symbolic data.

The following two results are direct consequences of the above definitions.

**Proposition 2.** *Let $\mathcal{L} = ((\Sigma, \Pi, Cfg), \mathcal{T}, \mathcal{S})$ be a language definition and let $\mathcal{L}^{\mathfrak{s}} = ((\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, Cfg^{\mathfrak{s}}), \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ be a symbolic extension of $\mathcal{L}$. Then for every concrete execution $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n \cdots$ and every symbolic configuration $\pi_0 \wedge^{\mathfrak{s}} \phi_0 \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$ such that $\gamma_0 \in [\![\pi_0 \wedge^{\mathfrak{s}} \phi_0]\!]$, there is a symbolic execution $\pi_0 \wedge^{\mathfrak{s}} \phi_0 \overset{\alpha^{\mathfrak{s}}_1}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge^{\mathfrak{s}} \phi_1 \overset{\alpha^{\mathfrak{s}}_2}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \overset{\alpha^{\mathfrak{s}}_n}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge^{\mathfrak{s}} \phi_n \cdots$ such that $\gamma_i \in [\![\pi_i \wedge^{\mathfrak{s}} \phi_i]\!]$ for $i = 0, 1, \ldots$.*

A symbolic execution $\pi_0 \wedge^{\mathfrak{s}} \phi_0 \overset{\alpha^{\mathfrak{s}}_1}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge^{\mathfrak{s}} \phi_1 \overset{\alpha^{\mathfrak{s}}_2}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots$ is **feasible** iff each side-condition $\phi_i \in \mathcal{T}^{\mathfrak{s}}_{sat}$, i.e., $\phi_i$ (seen as a FOL formula) is satisfiable in $\mathcal{T}$, for $i = 0, 1, \ldots$.

**Proposition 3.** *Let $\mathcal{L} = ((\Sigma, \Pi, Cfg), \mathcal{T}, \mathcal{S})$ be a language definition and let $\mathcal{L}^{\mathfrak{s}} = ((\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, Cfg^{\mathfrak{s}}), \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ be a symbolic extension of $\mathcal{L}$. Then for every feasible symbolic execution $\pi_0 \wedge^{\mathfrak{s}} \phi_0 \overset{\alpha^{\mathfrak{s}}_1}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge^{\mathfrak{s}} \phi_1 \overset{\alpha^{\mathfrak{s}}_2}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \overset{\alpha^{\mathfrak{s}}_n}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge^{\mathfrak{s}} \phi_n \cdots$ there is a concrete execution $\gamma_0 \overset{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \overset{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \overset{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n \cdots$ such that $\gamma_i \in [\![\pi_i \wedge^{\mathfrak{s}} \phi_i]\!]$ for $i = 0, 1, \ldots$.*

The next definition is essential for the definition of symbolic extensions, specifically, for the coverage/precision constraints that symbolic extensions have to satisfy.

**Definition 12.** Two terms $t, t'$ are **concretely unifiable** if there exists a valuation $\rho : Var \to \mathcal{T}$ such that $\rho(t) = \rho(t')$.

We now give sufficient conditions to obtain a symbolic extension of a given language according to Definition 11:

**Assumption 1.**
1. The rules $\mathcal{S}$ have the form $\pi_1 \wedge \phi_1 \Rightarrow \pi_2$, with $\pi_1$ a linear and "data-abstract" term, i.e. all subterms of data sort are variables, $\phi_1$ a quantifier-free FOL formula, such that $var(\pi_2, \phi_1) \subseteq var(\pi_1)$ and $var(\pi_1) \cap Var^{\mathfrak{d}} = \emptyset$. The left-hand side $\pi_1$ may have variables of data sort from $Var \setminus Var^{\mathfrak{d}}$.
2. $\mathcal{S}^{\mathfrak{s}} = \{(\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge sat(\psi \wedge \phi_1) \Rightarrow \pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1) \mid \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \in \mathcal{S}\}$, where $\psi$ is a fresh variable of sort *Bool*.

3. For all rules $\pi_1 \wedge \phi_1 \Rightarrow \pi_2$ in $\mathcal{S}$ and $\pi \wedge^{\mathfrak{s}} \phi \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$, $\pi_1$ and $\pi$ are concretely unifiable iff there exists a symbolic valuation $\sigma : Var \to \mathcal{T}^{\mathfrak{s}}$ such that $\sigma(\pi_1) = \pi$ and for each concrete unifier $\rho : Var \to \mathcal{T}$ there exists a valuation $\eta : Var^{\mathfrak{d}} \to \mathcal{T}$ such that $\rho = \eta \circ \sigma$.

This assumptions require some explanations. The first point is required for ensuring that semantical rules in $\mathcal{S}$ can be mapped to rewrite theories and that syntactical unification can be implemented by matching. Note that the linearity and data-abstract nature of the pattern can always be obtained by replacing duplicate variables and non-data subterms by fresh variables and by equating the fresh variables in the condition to what they replaced. The second point can be seen as the definition of the set of rules of the symbolic extension of a language definition. The third point actually says that matching can be seen as a mechanism for computing all concrete unifiers of two terms (this is established ahead in the paper, Theorem 3, Page 24).

**Theorem 1** ($\mathcal{L}^{\mathfrak{s}}$ is a symbolic extension of $\mathcal{L}$)**.** *Under Assumption 1 on the semantical rules $\mathcal{S}$ and $\mathcal{S}^{\mathfrak{s}}$, the language definition $\mathcal{L}^{\mathfrak{s}}$ whose semantic domain is defined as in Definition 11 is a symbolic extension of $\mathcal{L}$.*

## 5. Symbolic Extensions of Rewrite Theories

We define a notion of a symbolic extension of a rewrite theory, which translates the notions of symbolic execution of programs (symbolic values, path condition, relations with concrete execution) at the rewrite-theory level.

The definition of symbolic extension has two parts. The first one, called *symbolic pre-extension*, captures the syntactical features of symbolic execution, whereas the second part captures the corresponding semantical features.

**Definition 13** (Symbolic pre-extension of rewrite theory)**.** Consider an executable rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R, \vdash)$ with a distinguished sort *Cfg* such that $\mathcal{R}$ is topmost w.r.t. *Cfg*. Consider a countable number of symbolic values $V^{\mathfrak{s}}$, each of which has a sort in $\Sigma$. We say that an executable rewrite theory $\mathcal{R}^{\mathfrak{s}} = (\Sigma^{\mathfrak{s}}, E^{\mathfrak{s}} \cup A^{\mathfrak{s}}, R^{\mathfrak{s}}, \vdash^{\mathfrak{s}})$ is a **symbolic pre-extension** of $\mathcal{R}$ whenever:

- $\Sigma(V^{\mathfrak{s}}) \subset \Sigma^{\mathfrak{s}}$;
- $\Sigma^{\mathfrak{s}}$ contains a new sort $Cfg^{\mathfrak{s}}$ with constructor $\_\wedge^{\mathfrak{s}}\_ : Cfg \times Bool \to Cfg^{\mathfrak{s}}$;

- $E \subseteq E^{\mathfrak{s}}$, $A \subseteq A^{\mathfrak{s}}$;
- there is a bijection $\_^{\mathfrak{s}} : R \to R^{\mathfrak{s}}$;
- $\mathcal{R}^{\mathfrak{s}}$ is topmost w.r.t. $Cfg^{\mathfrak{s}}$.

Every $\gamma \in Can_{\Sigma,E/A,Cfg}$ is called a **configuration**, and every $\pi \wedge^{\mathfrak{s}} \phi \in Can_{\Sigma^{\mathfrak{s}},E^{\mathfrak{s}}/A^{\mathfrak{s}},Cfg^{\mathfrak{s}}}$ (often denoted by $\varphi, \varphi', \ldots$) is called **symbolic configuration**. These notions/notations are obviously borrowed from language definitions.

**Definition 14** (Satisfaction relation)**.** For any configuration $\gamma$, symbolic configuration $\pi \wedge^{\mathfrak{s}} \phi$, and substitution $\rho : V^{\mathfrak{s}} \to Can_{\Sigma,E/A}$ mapping each symbolic value to a canonical form of a term of corresponding sort, we write $(\gamma, \rho) \models \pi \wedge^{\mathfrak{s}} \phi$ whenever $\gamma =_{E \cup A} \rho(\pi)$ and $\vdash \rho(\phi)$ holds. We denote by $[\![\pi \wedge^{\mathfrak{s}} \phi]\!]$ the set of configurations $\{\gamma \in Can_{\Sigma,E/A,Cfg} \mid (\exists \rho)\ s.t.\ (\gamma, \rho) \models \pi \wedge^{\mathfrak{s}} \phi\}$.

The notion of symbolic extension requires, in addition to the syntactical requirement posed by symbolic pre-extension, that the concrete and symbolic transition relations satisfy the coverage and precision properties:

**Definition 15** (Symbolic extension of a rewrite theory)**.** A rewrite theory $\mathcal{R}^{\mathfrak{s}} = (\Sigma^{\mathfrak{s}}, E^{\mathfrak{s}} \cup A^{\mathfrak{s}}, R^{\mathfrak{s}}, \vdash^{\mathfrak{s}})$ is a symbolic extension of $\mathcal{R} = (\Sigma, E \cup A, R, \vdash)$ if $\mathcal{R}^{\mathfrak{s}}$ is a symbolic pre-extension of $\mathcal{R}$ and moreover:

> **Coverage:** if $\gamma \in [\![\varphi]\!]$ and $\gamma \xrightarrow{\alpha}_{\mathcal{R}} \gamma'$ then there exists a symbolic configuration $\varphi'$ such that $\gamma' \in [\![\varphi']\!]$ and $\varphi \xrightarrow{\alpha^{\mathfrak{s}}}_{\mathcal{R}^{\mathfrak{s}}} \varphi'$;
>
> **Precision:** if $\gamma' \in [\![\varphi']\!]$ and $\varphi \xrightarrow{\alpha^{\mathfrak{s}}}_{\mathcal{R}^{\mathfrak{s}}} \varphi'$ then there exists a configuration $\gamma$ such that $\gamma \xrightarrow{\alpha}_{\mathcal{R}} \gamma'$.

## 6. Faithful Encodings

In this section we present the faithful encodings of language definitions as rewrite theories (with entailment). We prove that the notions of faithful encoding and of symbolic extension commute, i.e., the faithful encoding $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ of a symbolic extension $\mathcal{L}^{\mathfrak{s}}$ of a concrete language $\mathcal{L}$ is a symbolic extension, in terms of rewrite theories, of the encoding $\mathcal{R}(\mathcal{L})$ of the concrete language $\mathcal{L}$; this, in a nutshell, are the results described in the diagram of the left-hand side of Figure 1.

**Definition 16.** Let $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ be a language definition that satisfies Assumption 1.1, and $\mathcal{R}(\mathcal{L}) = (\Sigma', E \cup A, R, \vdash)$ a rewrite theory such that

1. for each $\Sigma$-sort $s$, there is an inclusion map $\mathcal{T}_s \hookrightarrow T_{\Sigma', E \cup A, s}$,
2. there is a bijection $\alpha \mapsto \alpha'$ between $\mathcal{S}$ and $R$.

Then $\mathcal{R}(\mathcal{L})$ is a **faithful encoding** of $\mathcal{L}$ iff the following property holds: $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ iff $\gamma \overset{\alpha'}{\longrightarrow}_{\mathcal{R}(\mathcal{L})} \gamma'$, for all concrete configurations $\gamma, \gamma' \in \mathcal{T}_{\mathit{Cfg}}$.

Note that $\gamma$ and $\gamma'$ in $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ are elements in $\mathcal{T}$ and, by the inclusion morphism, in $\gamma \overset{\alpha'}{\longrightarrow}_{\mathcal{R}(\mathcal{L})} \gamma'$ are equivalence classes in $T_{\Sigma', E \cup A, s}$. Even if the definition let some freedom in defining the bijection between rules, we may think that the rules in $\mathcal{S}$, which by Assumption 1 are of the form $\pi \wedge \phi \Rightarrow \pi'$, are in correspondence with rewrite rules $\pi \to \pi'$ **if** $\phi$.

Faithful encodings enjoy the coverage and precision results (as stated in Definition 15), which ensure that both positive and negative outcomes of reachability analysis performed on faithfully encoding theories (e.g., performed by the Maude `search` command) hold on the language definitions.

**Assumption 2.**

1. $\Pi \setminus \Pi^{\mathfrak{d}} = \emptyset$ (i.e. there are no non-data predicates).
2. The result sort for any function symbol $f$ in $\Sigma \setminus \Sigma^{\mathfrak{d}}$ is not a $\Sigma^{\mathfrak{d}}$-sort.
3. Some function symbols in $\Sigma \setminus \Sigma^{\mathfrak{d}}$ are the subject of axioms like associativity, commutativity, unity, .... Let $A$ be the set of all these axioms. We assume that the axioms $A$ are *linear*, *regular*, and *data collapse-free*. A term $t$ is **linear** iff any variable occurs in $t$ at most once, an $u = v$ is **regular** iff $\mathit{var}(u) = \mathit{var}(v)$ and it is **linear** if both sides $u$ and $v$ are linear. An axiom $u = v$ is **data collapse-free** iff it does not collapse a non-data term into a data term; formally, for any substitution $\sigma$ neither $\sigma(u)$ nor $\sigma(v)$ is a variable of data sort. We also asume that there is a matching algorithm modulo $A$.
4. $\mathcal{T} = \mathcal{D}_!^{(\Sigma,\Pi)}/=_A$ and it is decidable whether $(\mathcal{T}_{t_1}, \ldots \mathcal{T}_{t_n}) \in \mathcal{T}_p$, for all $p \in \Pi$ and $t_1, \ldots, t_n$ $E/A$ canonical forms.

The assumptions 2.1 and 2.2 are just for the sake of presentation. In Remark 2 we show hints how the non-data predicates and data functions with non-data arguments can be encoded in the associated rewrite theory. Associativity and/or commutativity is needed for some structures used in the definition of the semantics. For instance, the order of the cells in the definition of the configuration is not fixed. This is essential for the modularity:

one may modify the configuration without touching the existing semantical rules. Recall that a configuration constructor is non-data (i.e. not in $\Sigma^{\mathfrak{d}}$). The associativity of the $\curvearrowright$-lists populating the k-cell is also important for modularity.

**Definition 17.** Let $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ be a language definition that satisfies Assumptions 1.1 and 2. Then the **rewrite theory** $\mathcal{R}(\mathcal{L}) = (\Sigma \cup \Sigma^{\Pi}, E \cup A, R, \vdash)$ is defined as follows:

- $\Sigma^{\Pi}$ contains a new sort $\mathit{Bool}_{\mathcal{L}}$[3], with constants $\mathit{true}_{\mathcal{L}}, \mathit{false}_{\mathcal{L}}$, propositional operations $(\neg_{\mathcal{L}}, \wedge_{\mathcal{L}})$, and one operation $p : s_1 \dots s_m \to \mathit{Bool}_{\mathcal{L}}$ for each predicate $p \in \Pi_{s_1 \dots, s_m}$;
- $A$ is the set of axioms from Assumption 2.3;
- for each operation $f$ in $\Sigma^{\mathfrak{d}}$ and values $d_1, \dots, d_n \in \mathcal{D}$ of the corresponding sorts, $E$ includes an equation $f(d_1, \dots, d_n) = \mathcal{D}_f(d_1, \dots, d_n)$.
- $R = \mathcal{S}$, where each rule $(\pi_1 \wedge \phi_1 \Rightarrow \pi_2) \in \mathcal{S}$ becomes a rewrite rule $(\pi_1 \to \pi_2 \,\mathbf{if}\, \phi_1) \in R$, where $\phi_1$ is a term of sort $\mathit{Bool}_{\mathcal{L}}$;
- $\vdash$ is defined such that
  - $\vdash p(t_1, \dots t_n)$ iff $p(t_1, \dots t_n) \to^{!}_{E/A} p(t'_1, \dots t'_n)$ and $(\mathcal{T}_{t'_1}, \dots \mathcal{T}_{t'_n}) \in \mathcal{T}_p$,
  - $\vdash \neg_{\mathcal{L}} \phi$ iff $\nvdash \phi$, and
  - $\vdash (\phi_1 \wedge_{\mathcal{L}} \phi_2)$ iff $\vdash \phi_1$ and $\vdash \phi_2$

  for all predicates $p \in \Pi$ and for all quantifier-free FOL formulas $\phi, \phi_1, \phi_2$.

Since $\to_{E/A}$ is confluent and terminating (see below), it follows that $\vdash$ is well defined. Note that $p(t_1, \dots t_n) \to^{!}_{E/A} p(t'_1, \dots t'_n)$ iff $t_i \to^{!}_{E/A} t'_i$, for $i = 1, \dots, n$.

**Theorem 2.** *Let $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ be a language definition that satisfies Assumptions 1.1 and 2. Then $\mathcal{R}(\mathcal{L})$ defined as in Definition 17 is an executable rewrite theory that faithfully encodes $\mathcal{L}$.*

[3]For technical reasons, the sort $\mathit{Bool}_{\mathcal{L}}$ is specific to the $\mathcal{R}(\mathcal{L})$ encoding of the language $\mathcal{L}$, and it is distinct from, e.g., a Boolean sort that may exist in the language's syntax $\Sigma$.

We shall prove that Assumption 1.3, regarding the relationship between the concretely unifiable configurations and matching symbolic configurations, is a consequence of the other assumptions.

**Theorem 3** (Unification by Matching)**.** *Let $\mathcal{L}$ and $\mathcal{L}^{\mathfrak{s}}$ be language definitions satisfying Assumptions 1.1,1.2 and 2. For all rules $\pi_1 \wedge \phi_1 \Rightarrow \pi_2$ in $\mathcal{S}$ and $\pi \in \mathcal{T}^{\mathfrak{s}}_{Cfg}$, $\pi_1$ and $\pi$ are concretely unifiable if and only if $match_A(\pi_1, \pi) \triangleq \{\sigma : var(\pi_1) \to T_\Sigma(Var^{\mathfrak{d}}) \mid \sigma(\pi_1) =_A \pi\} \neq \emptyset$ and for each concrete unifier $\rho$ of $\pi_1$ and $\pi$ there is $\sigma \in match_A(\pi_1, \pi)$ and a valuation $\eta$ such that $\rho = \eta \circ \sigma$.*

The result stated by the first part of Theorem 3 is similar to the Matching Lemma in [4]. The second part, which says that any concrete unifier can be written as the composition of a matcher and a valuation of data variables, is essential for proving the coverage and precision properties (see Theorem 1).

**Corollary 1.** *Let $\mathcal{L} = ((\Sigma, \Pi, Cfg), \mathcal{T}, \mathcal{S})$ and $\mathcal{L}^{\mathfrak{s}} = ((\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, Cfg^{\mathfrak{s}}), \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ be language definitions satisfying Assumptions 1.1, 1.2 and where the semantic domain $\mathcal{T}^{\mathfrak{s}}$ is defined like in Definition 11. Then $\mathcal{L}^{\mathfrak{s}}$ is a symbolic extension of $\mathcal{L}$.*

Definition 11 leaves some freedom in defining a symbolic extension. The following particular definition preserves the symbolic extension property for the encoding rewrite theories.

**Definition 18.** The precise definition for the semantic domain $\mathcal{T}^{\mathfrak{s}}$ is as follows:

- Any $\Sigma^{\mathfrak{s}}$-sort $s$ is interpreted as the set of $E$-canonical forms $Can_{\Sigma^{\mathfrak{s}},E,s}$;
- $\mathcal{T}^{\mathfrak{s}}_f(t_1, \ldots, t_n)$ is the $E$-canonical form of $f(t_1, \ldots, t_n)$.

The following corollary summarizes the results (graphically depicted in the left-hand side of Figure 1).

**Corollary 2.** *If $\mathcal{L}$ satisfies Assumption 2 and $\mathcal{L}^{\mathfrak{s}}$ is defined as in Definition 18 then $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ is a symbolic extension of $\mathcal{R}(\mathcal{L})$.*

*Proof.* We apply Theorem 2 for both $\mathcal{L}$ and $\mathcal{L}^{\mathfrak{s}}$ and obtain that $\mathcal{R}(\mathcal{L})$ faithfully encodes $\mathcal{L}$ and $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ faithfully encodes $\mathcal{L}^{\mathfrak{s}}$. Since $\mathcal{L}^{\mathfrak{s}}$ is a symbolic extension of $\mathcal{L}$ the transition systems of the two languages are related by the coverage and precision results. Since faithful encoding produces an executable rewrite theory whose transition system is isomorphic to the language

being encoded, the transition systems of the (executable) rewrite theories $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ and of $\mathcal{R}(\mathcal{L})$ also satisfy the coverage and precision results. The syntactical requirements for $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ to be a (pre)symbolic extension of $\mathcal{R}(\mathcal{L})$ also follow from the definitions, which concludes the proof of this corollary. □

*Remark* 2. The constraints given by Assumptions 2.1 (no non-data predicates) and 2.2 (no functions in $\Sigma \setminus \Sigma^{\mathfrak{d}}$ that return data results) can be relaxed. We claim that all results reported in this section hold if the languages definition include such predicates and functions having the following properties.

Assume that for each predicate $p \in \Pi \setminus \Pi^{\mathfrak{d}}$ the definition of $p$ in $\mathcal{T}$ can be encoded with a set of confluent and terminating equations $E^p$ that are consistent with data. Let $E^{\Pi}$ be the union of these sets of equations. A simple example of such a predicate is `WellTyped(`$E$`)` that holds whenever $E$ is a well typed expression; this predicate can be easily specified by structural induction following the syntax definition of the expressions. A predicate $p$ is *consistent with data* if it does not depend on the term representation of data, i.e., if $t =_{E^{\mathfrak{d}}} t'$ then $\vdash p(\ldots t \ldots)$ iff $\vdash p(\ldots t' \ldots)$, where $E^{\mathfrak{d}}$ denote the set of equations given by Definition 17. For instance, a predicate `foo` satisfying

$$\begin{array}{ll} \vdash \texttt{foo}(\ldots I \ldots) & \text{iff } I \text{ is an odd integer} \\ \vdash \texttt{foo}(\ldots t_1 +_{Int} t_2 \ldots) & \text{iff } \vdash \texttt{foo}(\ldots t_1 \ldots) \wedge \vdash \texttt{foo}(\ldots t_2 \ldots) \end{array}$$

is not consistent with data because $\vdash \texttt{foo}(\ldots 3 +_{Int} 5 \ldots)$ and $\nvdash \texttt{foo}(\ldots 8 \ldots)$.

Similarly, we assume that $\Sigma \setminus \Sigma^{\mathfrak{d}}$ includes a sub-signature $\Sigma^{\mathfrak{f}}$ such that for each function symbol $f \in \Sigma^{\mathfrak{f}} \setminus \Sigma^{\mathfrak{d}}$ the definition of $f$ in $\mathcal{T}$ can be encoded with a set of confluent and terminating equations $E^f$ that preserve data. Let $E^{\mathfrak{f}}$ be the union of these sets of equations. A function $f$ preserves data iff $t =_{E^{\mathfrak{d}}} t'$ implies $f(\ldots t \ldots) =_{E^{\mathfrak{f}}} (\ldots t' \ldots)$. An example of such function is `typedName()` that returns the type of a name in a C++ declaration; e.g., `typedName(int* f(bool))` returns `"f of function (bool) returning pointer to int"`. Such a function preserves data and can be defined by structural induction following the syntax definition of the declarations. We further assume that $(E^{\mathfrak{d}} \cup E^{\Pi} \cup E^{\mathfrak{f}})/A$ is ground confluent, terminating, and coherent as well. Then we may consider $\mathcal{T} = \mathcal{D}\!\upharpoonright^{(\Sigma \setminus \Sigma^{\mathfrak{f}}, \Pi)}\!/=_A$, $E = E^{\mathfrak{d}} \cup E^{\mathfrak{f}}$, change the definition of $\vdash$ such that

- if $p \in \Pi^{\mathfrak{d}}$:
  $\vdash p(t_1, \ldots t_n)$ iff $p(t_1, \ldots t_n) \rightarrow^{!}_{E/A} p(t'_1, \ldots t'_n)$ and $(\mathcal{T}_{t'_1}, \ldots \mathcal{T}_{t'_n}) \in \mathcal{T}_p$,

- if $p \in \Pi \setminus \Pi^{\mathfrak{d}}$:
  $\vdash p(t_1, \ldots t_n)$ iff $p(t_1, \ldots t_n) \rightarrow^{!}_{(E \cup E^{\Pi})/A} \mathit{True}$.

The equations $E^{\mathfrak{f}}$ have the same properties like $E^{\mathfrak{d}}$ in the above proofs and the new definition of $\vdash$ faithfully encodes the definitions of the predicates. Note that the terms representing elements in $\mathcal{T}$ do not includes function symbols in $\Sigma^{\mathfrak{f}}$ because they are $E^{\mathfrak{f}}$-normal forms. We have avoided to consider these additional functions and predicates in order to keep the presentation as simple as possible.

## 7. Approximate Encodings

When attempting to verify, analyze, or symbolically execute programs, one is usually faced with a natural state-explosion problem. This arises either from language nondeterminism, in the case of exploring the state space of nondeterministic or concurrent systems, or from nondeterminism given by abstracting the input, in the case of symbolic execution.

To cope with this explosion problem, one usually employs various abstractions to achieve approximate models of the intended one, be them either over-approximations —when one attempts to ensure a property over a larger model—, or under-approximations —when one looks for errors and wants to maintain soundness to ensure that if a error is found, then there is indeed an error of the intended model and not in the approximation.

Our approximate encodings fall into the under-approximation class; the goal here is to achieve a good balance between the depth of abstraction (which reduces the search space) and the precision of the analysis (which is reduced by the abstraction).

There are currently two main proposals in the literature for obtaining abstractions of the rewrite theories: equational abstractions [14] or transforming some semantical rules into equations [15]. The former amounts to essentially deriving a new definition, where the new model $\mathcal{T}$ is the quotient of the original one, typically requiring substantial input from the user, which is something we would like to avoid because, usually, the equations depend on programs, not on languages.

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 analyse 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 be 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 and/or termination, 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 19.** A **two-layered rewrite theory** is a tuple $\mathcal{R} = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash, \varepsilon)$, where $(\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash)$ is an executable rewrite theory with entailment, $E \cup \mathit{1R}$ 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 $(E \cup \mathit{1R})/A$-irreducible terms $\{t' \in T_\Sigma \mid t \to^{!}_{(E \cup \mathit{1R})/A} t'\}$ (which is nonempty precisely because $E \cup \mathit{1R}$ is ground terminating modulo $A$). The one-step rewrite relation $\twoheadrightarrow_{\mathcal{R}}$ is defined by $t_1 \twoheadrightarrow_{\mathcal{R}} t_2$ iff $\varepsilon(t_1) \to_{\mathit{2R}/A} t_2'$ and $\mathit{can}_{E/A}(t_2') =_A t_2$.

If a language definition is implemented by a two-layered rewrite theory, then we may think that the executions of the programs are achieved by *representatives*. If $t$ represents the current state of the program then the set of all *1R*-executions starting from $t$ is finite and all these executions are finite; let $\mathit{enabled}_{\mathit{1R}}(t)$ denote the set of these executions. The function $\varepsilon$ chooses exactly one of these executions, say $\mathit{sample}_{\varepsilon}(t)$. So, the transition system defined by a two-layered rewrite theory $\mathfrak{R}(\mathcal{L})$ implementing $\mathcal{L}$ is an under-approximation of the one giving semantics to $\mathcal{L}$. Therefore, two-layered rewrite theories are different from equational abstractions, which produce an over-approximation. When a program is checked against a temporal property, its states are labelled with atomic properties. A rewrite rule $r$ is *invisible* if for any execution step $t_1 \to t_2$ obtained by applying $r$, $t_1$ and $t_2$ have the same labels (i.e. satisfy the same state predicates). If all rules in *1R* are invisible, the set $E \cup A \cup \mathit{1R}$ is confluent and *2R* is coherent with respect to $E \cup A \cup \mathit{1R}$, then the set *1R* in the definition of two-layered rewrite theories coincides with invisible transitions in standard rewrite theories presented in [15].

Examples of two-layered rewrite theories are shown in Section 8.

**Theorem 4.** *Let* $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ *be a language definition and* $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash, \varepsilon)$ *be a two-layered rewrite theory with* $(\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R})$ *built as in Definition 17 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* $\mathcal{L}$.

**Corollary 3** (precision for approximate encoding)**.** *Let* $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ *be a language definition and* $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash, \varepsilon)$ *be an approximate encoding of* $\mathcal{L}^{\mathfrak{s}}$. *For each feasible symbolic execution* $\pi_0 \wedge \phi_0 \twoheadrightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_1 \wedge \phi_1 \twoheadrightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots \twoheadrightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge \phi_n \twoheadrightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \cdots$ *there is a concrete execution in* $\mathcal{L}$: $\gamma_0 \Longrightarrow^{+}_{\mathcal{S}} \gamma_1 \Longrightarrow^{+}_{\mathcal{S}} \cdots \Longrightarrow^{+}_{\mathcal{S}} \gamma_n \Longrightarrow^{+}_{\mathcal{S}} \cdots$ *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 and precision relationships between $\mathcal{L}$ and $\mathcal{L}^{\mathfrak{s}}$ can be reflected on 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 build $\mathit{1R}^{\mathfrak{s}}$ from *1R* and $\mathit{2R}^{\mathfrak{s}}$ from *2R*, but this is not enough to have a coverage-like result. For example, the program `log` in Figure 5 is deterministic and terminating for each $\vartheta(A) \in \mathit{Int}$. Thus, one may execute any instance of it with an approximate encoding having no second-layer rules, i.e., $\mathit{2R} = \emptyset$. If $\mathit{2R}^{\mathfrak{s}} = \emptyset$, then $\mathit{1R}^{\mathfrak{s}}$ could be 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 than zero. Another problem is to specify how the strategy $\varepsilon$ is extended to $\varepsilon^{\mathfrak{s}}$. Since it is hard to give general answers for these questions we opted for a particular solution that can be implemented in Maude.

**Definition 20** (symbolic approximate encoding)**.** Let $\mathcal{L}^{\mathfrak{s}} = ((\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, \mathit{Cfg}^{\mathfrak{s}}), \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$ be the symbolic extension of $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ and $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash, \varepsilon)$ an approximate encoding of $\mathcal{L}$. We assume that there is a total order relation $\prec$ over *1R* such that:

1. the rewrite $t \rightarrow^{!}_{(E \cup \mathit{1R})/A} \varepsilon(t)$ uses the minimal applicable rule from *1R* w.r.t. $\prec$ whenever such a rule exists;
2. if $\alpha$ is unconditional and $\alpha'$ is conditional then $\alpha \prec \alpha'$.

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

- $\mathit{1R}^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{1R}, \alpha \text{ unconditional}\}$;
- $\mathit{2R}^{\mathfrak{s}} = \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{1R}, \alpha \text{ conditional}\} \cup \{\alpha^{\mathfrak{s}} \mid \alpha \in \mathit{2R}\}$;
- $\vdash^{\mathfrak{s}}$ the same with that of $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$;
- $\alpha^{\mathfrak{s}} \prec^{\mathfrak{s}} \alpha'^{\mathfrak{s}}$ only if $\alpha \prec \alpha'$;
- $\varepsilon^{\mathfrak{s}}$ uses the minimal rule from $\mathit{1R}^{\mathfrak{s}}$ w.r.t. $\prec^{\mathfrak{s}}$.

The partial order $\prec^{\mathfrak{s}}$ together with $\varepsilon$ and $\varepsilon^{\mathfrak{s}}$ are used to control the application order of the rules in the two theories, $\mathfrak{R}(\mathcal{L})$ and $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$. If a rewrite engine, like Maude, executes the equations and respectively the rules in the order they appear in the rewrite theory, then $\prec^{\mathfrak{s}}$ could be this order. A symbolic configuration represents a set of concrete configurations. If a conditional rule is applicable for a given symbolic configuration, then the condition could hold for some concrete configurations and not for others. Hence the symbolic execution must be branched in two sub-executions according to the two cases. In order to capture these behaviours the transitions given by these rules cannot be collapsed and therefore all conditional rules are included in $\mathit{2R}^{\mathfrak{s}}$.

The next result proves a coverage result between $\mathfrak{R}(\mathcal{L})$ and $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$, where the step by step simulation is replaced by a kind of stuttering simulation, which is close to that required by Definition 15.

**Theorem 5** (coverage for approximate encoding rewrite theories)**.** *Let* $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ *be a language definition,* $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash, \varepsilon)$ *be an approximate encoding of* $\mathcal{L}$*, and* $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ *be the approximate encoding of* $\mathcal{L}^{\mathfrak{s}}$ *defined as in Definition 20. If* $\gamma \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$, $\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$ *then there is* $\pi' \wedge^{\mathfrak{s}} \phi'$ *such that* $\pi \wedge^{\mathfrak{s}} \phi \rightarrow^{+}_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi' \wedge^{\mathfrak{s}} \phi'$ *and* $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$.

A 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 $\varepsilon$. Therefore there are no guarantees that the concrete execution given by Corollary 3 will be the same with that chosen by the strategy $\varepsilon$.

If the strategies $\varepsilon^{\mathfrak{s}}$ are "isomorphic images" of $\varepsilon$, then we have a precision-like result.

**Theorem 6** (precision for approximate encoding rewrite theories)**.** *Let* $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ *be a language definition,* $\mathfrak{R}(\mathcal{L}) = (\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash, \varepsilon)$

*be an approximate encoding of* $\mathcal{L}$*, and* $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ *be the approximate encoding of* $\mathcal{L}^{\mathfrak{s}}$ *defined as in Definition 20 such that* $\varepsilon^{\mathfrak{s}}$ *is an isomorphic image of* $\varepsilon$ *via the transformation* $\bullet \mapsto \bullet^{\mathfrak{s}}$*. If* $\pi \wedge^{\mathfrak{s}} \phi \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi' \wedge^{\mathfrak{s}} \phi'$ *and* $\gamma' \models \pi' \wedge^{\mathfrak{s}} \phi'$ *then there exists* $\gamma$ *such that* $\gamma \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$ *and* $\gamma \models \pi \wedge^{\mathfrak{s}} \phi$*.*

The isomorphism between $\varepsilon$ and $\varepsilon^{\mathfrak{s}}$ can be easily obtained. We already have a 1-1 correspondence between $\mathcal{S}$ and $\mathcal{S}^{\mathfrak{s}}$ using the transformation given in [3]. If the rewrite engine tries to execute the equations/rules in the order they are written in the rewrite theory, then we take $\prec$ being defined by this order. The following example illustrates this feature.

*Example* 9. Let $P_0$ denote the CinK program `a = --y / x;` and $\gamma_0$ the configuration $\langle\langle P_0 \rangle_{\mathsf{k}} \langle E_0 \rangle_{\mathsf{env}} \langle S_0 \rangle_{\mathsf{store}} \cdots \rangle_{\mathsf{cfg}}$, where $E_0 \triangleq \mathtt{x} \mapsto \ell_x \; a \mapsto \ell_a \; \mathtt{y} \mapsto \ell_x$ and $S_0 \triangleq \ell_x \mapsto 1 \; \ell_a \mapsto 0$. This configuration can be obtained, e.g., by compiling the program `int x = 1, a = 0; int& y = x; a = -y / x;`. Recall that the evaluation order for the division operator is nondeterministic. Assume that the heating-cooling rules (1)-(4) are in *1R*. Their symbolic versions are:

$$A_1 \; / \; A_2 \wedge^{\mathfrak{s}} \psi \Rightarrow \mathit{rvalue}(A_1) \curvearrowright \Box \; / \; A_2 \wedge^{\mathfrak{s}} \psi \tag{5}$$

$$A_1 \; / \; A_2 \wedge^{\mathfrak{s}} \psi \Rightarrow \mathit{rvalue}(A_2) \curvearrowright A_1 \; / \; \Box \wedge^{\mathfrak{s}} \psi \tag{6}$$

$$\mathit{rvalue}(I_1) \curvearrowright \Box \; / \; A_2 \wedge^{\mathfrak{s}} \psi \Rightarrow I_1 \; / \; A_2 \wedge^{\mathfrak{s}} \psi \tag{7}$$

$$\mathit{rvalue}(I_2) \curvearrowright A_1 \; / \; \Box \wedge^{\mathfrak{s}} \psi \Rightarrow A_1 \; / \; I_2 \wedge^{\mathfrak{s}} \psi \tag{8}$$

If the strategy $\varepsilon$ chooses to apply first the rules (1) and (3), respectively, then we obtain the intermediate configuration $\gamma_1 \triangleq \langle\langle P_1 \rangle_{\mathsf{k}} \langle E_0 \rangle_{\mathsf{env}} \langle S_1 \rangle_{\mathsf{store}} \cdots \rangle_{\mathsf{cfg}}$ with $P_1$ equal to $\mathtt{0/x} \curvearrowright \ell_{\mathtt{a}} = \Box$; and $S_1$ the store $\ell_x \mapsto 0 \; \ell_a \mapsto 0$.

Let $\gamma_0^{\mathfrak{s}}$ denote the symbolic configuration $\langle\langle P_0 \rangle_{\mathsf{k}} \langle E_0 \rangle_{\mathsf{env}} \langle S_0^{\mathfrak{s}} \rangle_{\mathsf{store}} \cdots \rangle_{\mathsf{cfg}} \wedge^{\mathfrak{s}} x \geq 1$, with $S_0^{\mathfrak{s}} \triangleq \ell_x \mapsto x \; \ell_a \mapsto a$. If the strategy $\varepsilon^{\mathfrak{s}}$ chooses to apply first the rules (6) and (8), respectively, then we obtain the intermediate configuration $\gamma_2^{\mathfrak{s}} \triangleq \langle\langle P_2^{\mathfrak{s}} \rangle_{\mathsf{k}} \langle E_0 \rangle_{\mathsf{env}} \langle S_0^{\mathfrak{s}} \rangle_{\mathsf{store}} \cdots \rangle_{\mathsf{cfg}} \wedge^{\mathfrak{s}} x \geq 1$ with $P_2^{\mathfrak{s}}$ equal to `--y / x` $\curvearrowright \ell_{\mathtt{a}} = \Box$;. Obviously, the computation $\gamma_0^{\mathfrak{s}} \Rightarrow_{\mathcal{S}}^* \gamma_2^{\mathfrak{s}}$ does not cover $\gamma_0 \Rightarrow_{\mathcal{S}}^* \gamma_1$. But we get coverage if the strategy $\varepsilon^{\mathfrak{s}}$ chooses to apply first the rules (5) and (7), respectively, and this can be achieved by setting (1) $\prec$ (2). Moreover, we have precision as well because there is an isomorphism between $\varepsilon$ and $\varepsilon^{\mathfrak{s}}$.

Assume now that the rules (1)-(4) are replaced with the following condi-

tional ones:

$$A_1 \;/\; A_2 \wedge \neg A_1 :: \mathit{Int} \Rightarrow \mathit{rvalue}(A_1) \curvearrowright \square \;/\; A_2 \tag{9}$$

$$A_1 \;/\; A_2 \wedge \neg A_2 :: \mathit{Int} \Rightarrow \mathit{rvalue}(A_2) \curvearrowright A_1 \;/\; \square \tag{10}$$

$$\mathit{rvalue}(I_1) \curvearrowright \square \;/\; A_2 \wedge A_1 :: \mathit{Int} \Rightarrow I_1 \;/\; A_2 \tag{11}$$

$$\mathit{rvalue}(I_2) \curvearrowright A_1 \;/\; \square \wedge A_2 :: \mathit{Int} \Rightarrow A_1 \;/\; I_2 \tag{12}$$

where :: is the membership predicate. Then, their symbolic versions are in $\mathit{2R}^{\mathfrak{s}}$.

The computation $\gamma_0 \Rightarrow_{\mathcal{S}}^{*} \gamma_1$ is now covered by $\gamma_0^{\mathfrak{s}} \Rightarrow_{\mathcal{S}}^{*} \gamma_1^{\mathfrak{s}}$, where $\gamma_1^{\mathfrak{s}}$ is $\langle\langle P_1^{\mathfrak{s}}\rangle_{\mathsf{k}}\langle E_0\rangle_{\mathsf{env}}\langle S_1^{\mathfrak{s}}\rangle_{\mathsf{store}}\cdots\rangle_{\mathsf{cfg}} \wedge^{\mathfrak{s}} x \geq 1$ with $P_1^{\mathfrak{s}}$ equal to $x -_{\mathit{Int}} 1/\mathtt{x} \curvearrowright \ell_{\mathtt{a}} = \square$; and $S_1^{\mathfrak{s}}$ equal to $\ell_x \mapsto x -_{\mathit{Int}} 1\ \ell_a \mapsto a$. But in this case we loose the precision for $\gamma_0^{\mathfrak{s}} \Rightarrow_{\mathcal{S}}^{*} \gamma_2^{\mathfrak{s}}$. The precision result can be obtained if we include rules (9)-(12) in $\mathit{2R}$.

## 8. $\mathbb{K}$ definitions as Maude theories

This section discusses encoding the representation of both $\mathbb{K}$ semantic definitions and their symbolic extensions as two layered rewrite theories in the Maude rewrite engine.

The $\mathbb{K}$ framework (up to version 3.5) uses Maude as a rewrite engine. 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 set of $\mathbb{K}$ rules is compiled into a two-layered rewrite theory, which is then encoded into Maude by using equations for the first-layer and rewrite rules for the second-layer. Being optimized for generating interpreters, the default behavior of the $\mathbb{K}$ compiler is to translate all $\mathbb{K}$ rules into (conditional) equations (i.e. $\mathit{1R} = \mathcal{S}$ 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 resulting Maude module. To do so, the names of the rules defining *transitions* must be passed as an argument for the `--transition` option at compilation time:

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

The above command specifies that the rule tagged with the *division* attribute is a transition; thus, the rule for division is included in $2R$. By making it a rewrite rule in Maude, we can explore the nondeterminism generated by the rule when using Maude's `search` command.

A source of nondeterminism arises from strictness annotations. When the *strict* attribute is given to some syntactical construct, the tool chooses by default an arbitrary, but fixed order to evaluate its arguments. This optimisation has the side effect of possibly losing behaviours due to missed interleavings. Some of these interleavings can be restored using the `--superheat` option. This option is used to instruct the $\mathbb{K}$ tool to exhaustively explore all the nondeterministic evaluation choices for the strictness of a language construct.

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 5) will result in a single solution, where the returned value is either 1 (when the tool first evaluates `dec()` and then `inc()`) or 3 (when it evaluates `inc()` before `dec()`). However, if we set the operation *plus* as superheat:

```
$ kompile cink.k --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:

```
$ krun counter.cink --search
```

### *8.1. Representing the symbolic extensions*

The symbolic transformations discussed in Section 4.2 are implemented as compilation steps in the $\mathbb{K}$ compiler [3]. 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 property (Theorem 5). 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 [16] 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. To obtain $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$ from a $\mathbb{K}$ definition one uses the symbolic backend as follows:

```
$ kompile cink.k --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 or symbolic values. 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 final states to be searched using the `-bound` option, or the exploration depth using the `-depth` option (in the same way):

```
$ krun log.cink -cIN="ListItem(#symInt(a))" -cPC="true"
                                          \ --search --bound 3
```
[4]

This executes `log` (Figure 5) symbolically, until a number of 3 solutions is found. Each solution consists in a result configuration and a formula which constitutes the path condition. Users can also set the initial path condition using the `-cPC` option. This becomes useful when one wants to limit the search space by setting initial constraints over symbolic variables:

```
$ krun log.cink -cIN="ListItem(#symInt(a))" --search       \
         -cPC="0 <Int #symInt(a) andBool #symInt(a) <Int 10"
```

There are certain cases where setting initial constraints over symbolic values is important; for instance, in a program which contains two consecutive loops, the symbolic execution might get stuck in the first loop, and it never reaches the second one. In such cases it is a bit harder to control the number of iterations using only `--bound` and `--depth` options, since their behaviour depends on the number of rules in *2R*. On the other hand, when using constraints as in the command above, the tool returns the set of final solutions

[4] In the version 3.4 of the $\mathbb{K}$ Framework the internal representation of the symbolic variable $A$`:Int` is `ListItem(#symInt(a))`.

by exploring all possible values of the constrained symbolic variables, and thus, enabling the exploration of the second loop too. In fact, a generic solution to this problem would be to add counters for each loop and use the `-cPC` to set limits for them.

Since Z3 only approximates the predicate *sat*, our tool slices the paths for which *checkSat* returns "unsat". That is, it explores the feasible paths and some unfeasible paths, too (corresponding to the case when *checkSat* returns "unknown"). This might lead to a bigger search space, but the tool does not miss any feasible execution path. Note that coverage for all possible feasible paths of a program is obtained only when using faithful encodings.

## 9. Conclusion and Future Work

We have 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) analysis 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 faithful 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 their 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 analysis hold on which representations.

*Related Work*

The first tool supporting $\mathbb{K}$ [17] 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 used to generate code for various backends, including Maude. The tool and the semantics of $\mathbb{K}$ definitions are

described in [18]. The programming-language definition framework presented in this paper (Section 4) is a specialised case of that definition.

Rewriting modulo SMT was proposed as a new (language independent) technique which combines the power of SMT solving, rewriting modulo SMT, and model checking, for analysing infinite-state open systems [4]. An open system is modelled as a triple $(\Sigma, E, R)$, where $(\Sigma, E)$ is an equational theory describing the system states and $R$ is a set of rewrite rules describing the system's transitions. The state of an open system must include the state changes due to the environment. These changes are captured by new fresh symbolic variables introduced in the right-hand side of the rewrite rules. Thus, the system states are represented not as concrete states, but as symbolic ones, i.e. terms with variables (ranging in the domains handled by the SMT solver) which are constrained by an SMT-solvable formula. Rewriting modulo SMT can symbolically rewrite such states (which may describe possibly an infinite number of concrete states). Our coverage and precision properties, which relate the faithful rewrite-theory encoding of a language and of that language's symbolic version, are analogous to the soundness and completeness results in [4], which relate usual rewriting and rewriting modulo SMT. An interesting alternative to defining symbolic execution as executions in a transformed language (as we do it in [3]) would be to compile a language into a rewriting modulo SMT Maude module. Our construction of two-layered rewrite theories has some similarities with equational abstractions [14] and with the state-space reduction techniques obtained by transforming rules into equations presented in [15]. 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 [14, 15].

## References


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

[2] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, C. Talcott, All About Maude, A High-Performance Logical Framework, Vol. 4350 of LNCS, Springer, 2007.

[3] A. Arusoaie, D. Lucanu, V. Rusu, A generic framework for symbolic execution, in: 6th International Conference on Software Lan-

guage Engineering, Vol. 8225 of LNCS, Springer Verlag, 2013, pp. 281–301, also available as a technical report at http://hal.inria.fr/hal-00766220/.

[4] C. Rocha, J. Meseguer, C. A. Muñoz, Rewriting modulo SMT and open system analysis, in: Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers, 2014, pp. 247–262. doi:10.1007/978-3-319-12904-4_14.

[5] A. Arusoaie, D. Lucanu, V. Rusu, T.-F. Şerbănuţă, A. Ştefănescu, G. Roşu, Language Definitions as Rewrite Theories, in: 10th International Workshop on Rewriting Logic and Application, Grenoble, France, 2014, pp. 31–44, (To appear in Springer LNCS).

[6] D. Lucanu, T. F. Şerbănuţă, CinK - an exercise on how to think in K, Tech. Rep. TR 12-03, Version 2, Alexandru Ioan Cuza University, Faculty of Computer Science (December 2013).

[7] G. Roşu, A. Ştefănescu, Checking reachability using matching logic, in: G. T. Leavens, M. B. Dwyer (Eds.), OOPSLA, ACM, 2012, pp. 555–574.

[8] Working draft, standard for Programming Language C++, no. N3797.
URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf

[9] F. Durán, J. Meseguer, On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories, J. Log. Algebr. Program. 81 (7-8) (2012) 816–850.

[10] P. Viry, Equational rules for rewriting logic, Theor. Comput. Sci. 285 (2) (2002) 487–517.

[11] J. Meseguer, Software specification and verification in rewriting logic (2003).
URL http://maude.cs.uiuc.edu/papers/abstract/Mspec_ver_rwl_2003.html

[12] G. Rosu, Matching Logic - Extended Abstract (Invited Talk), in: 26th International Conference on Rewriting Techniques and Applications

(RTA 2015), Vol. 36 of Leibniz International Proceedings in Informatics (LIPIcs), 2015, pp. 5–21.

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

[14] J. Meseguer, M. Palomino, N. Martí-Oliet, Equational abstractions, Theor. Comput. Sci. 403 (2-3) (2008) 239–264.

[15] A. Farzan, J. Meseguer, State space reduction of rewrite theories using invisible transitions, in: Proceedings of the 21st German Annual Conference on Artificial Intelligence, Springer, 2006, pp. 142–157.

[16] L. M. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: TACAS, Vol. 4963 of Lecture Notes in Computer Science, Springer, 2008, pp. 337–340.

[17] T. F. Şerbanuţă, G. Roşu, K-Maude: A rewriting based tool for semantics of programming languages, in: P. C. Ölveczky (Ed.), Rewriting Logic and Its Applications - 8th International Workshop, Vol. 6381 of Lecture Notes in Computer Science, 2010, pp. 104–122.

[18] D. Lucanu, T. F. Şerbănuţă, G. Roşu, The K Framework distilled, in: 9th International Workshop on Rewriting Logic and its Applications, Vol. 7571 of Lecture Notes in Computer Science, Springer, 2012, pp. 31–53, invited talk.

## Proofs

PROOF OF THEOREM 1 (PAGE 20). We have to establish the coverage and precision properties required by the $\mathcal{L}^{\mathfrak{s}}$ construction (cf. Definition 11). We state and prove them as two lemmas, and the proof of our theorem is the direct consequence of them. The two lemmas are proved under the more general setting in which rules $\alpha$ may have elementary patterns in the right-hand side.

**Lemma 1** (Coverage)**.** *If* $\gamma \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ *and* $\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$ *with* $\pi \wedge^{\mathfrak{s}} \phi \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$*, then there exists* $\pi' \wedge^{\mathfrak{s}} \phi' \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$ *such that* $\pi \wedge^{\mathfrak{s}} \phi \overset{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$ *and* $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$.

*Proof.* Assume that $\alpha$ is $\pi_1 \wedge \phi_1 \twoheadrightarrow \pi_2 \wedge \phi_2$. From $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ we obtain $\rho : Var \to \mathcal{T}$ such that: (i) $\rho(\pi_1) = \gamma$, (ii) $\rho \models \phi_1$, and (iii) $\rho(\pi_2) = \gamma'$. From $\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$ we obtain $\rho' : Var \to \mathcal{T}$ such that (iv) $\rho'(\pi) = \gamma$, (v) $\rho' \models \phi$. Since we may assume $var(\pi \wedge \phi) \cap var(\pi_1 \wedge^{\mathfrak{s}} \phi_1) = \emptyset$ we may take $\rho = \rho'$. Thus, by (i) and (iv), $\rho$ is a concrete unifier of $\pi_1$ and $\pi$ and by Assumption 1 we obtain $\sigma^{\pi_1}_{\pi} : var(\pi_1) \to T_{\Sigma}(var(\pi))$, extended to $Var$ such that (vi) $\sigma^{\pi_1}_{\pi}(\pi_1) =_A \pi$,

---

Vlad:
de verificat $=_A$

---

(vii) $\sigma^{\pi_1}_{\pi}(x) = x$ for all $x \notin var(\pi_1)$, and $\eta : Var \to \mathcal{T}$ such that (viii) $\rho = \eta \circ \sigma^{\pi_1}_{\pi}$. Since we assumed $var(\pi \wedge \phi) \cap var(\pi_1 \wedge \phi_1) = \emptyset$, using (vii) we obtain that $\rho|_{var(\phi)} = \eta|_{var(\phi)}$, and from (v) we obtain (ix) $\eta \models \phi$. We now define $\sigma : Var \cup \{\psi\} \to \mathcal{T}^{\mathfrak{s}}$ as follows:

- $\sigma(x) = \sigma^{\pi_1}_{\pi}(x)$ if $x \in var(\pi_1)$;
- $\sigma(x) = \phi$ if $x = \psi$;
- $\sigma(x) = x$, otherwise.

We take $\pi' \triangleq \sigma(\pi_2)$ and $\phi' \triangleq \sigma(\psi \wedge \phi_1) = \phi \wedge \sigma(\phi_1)$, and show that the conclusions of the lemma hold. First, we note that $\pi' \wedge^{\mathfrak{s}} \phi' \in \mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$, since, by Assumption 1, $var(\pi_2) \subseteq var(\pi_1)$ and thus $var(\pi') = var(\sigma(\pi_2)) = var(\sigma^{\pi_1}_{\pi}(\pi_2)) \subseteq var(ran(\sigma^{\pi_1}_{\pi})) \subseteq Var^{\mathfrak{d}}$, and $var(\phi') = var(\phi) \cup \sigma^{\pi_1}_{\pi}(\phi_1) \subseteq Var^{\mathfrak{d}} \cup var(ran(\sigma^{\pi_1}_{\pi})) \subseteq Var^{\mathfrak{d}}$.

Proving $\pi \wedge^{\mathfrak{s}} \phi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$: From (ii) $\rho \models \phi_1$ and (viii) $\rho = \eta \circ \sigma^{\pi_1}_{\pi}$ we obtain $\eta \models \sigma^{\pi_1}_{\pi}(\phi_1) = \sigma(\phi_1)$. From the above and (ix): $\eta \models \phi$ we obtain

(x) $\eta \models (\phi \wedge \sigma(\phi_1)) = \sigma(\psi \wedge \phi_1)$, thus, $\sigma(\psi \wedge \phi_1)$ is satisfiable, meaning $sat(\sigma(\psi \wedge \phi_1))$ holds. The latter is equivalent to (xi) $\sigma(sat(\psi \wedge \phi_1))$ holds.

We also have (xii) $\sigma(\pi_1 \wedge^{\mathfrak{s}} \psi) = \pi \wedge^{\mathfrak{s}} \phi$. Using (xi), (xii) and $\pi' = \sigma(\pi_2)$, $\phi' = \phi \wedge \sigma(\phi_1)$ we obtain $\pi \wedge^{\mathfrak{s}} \phi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$.

Proving $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$: we have $\gamma' = \rho(\pi_2) = \eta \circ \sigma_{\pi}^{\pi_1}(\pi_2) = \eta \circ \sigma(\pi_2)$, thus, (xiii) $\gamma' = \eta(\pi')$, and using (x) $\eta \models \phi \wedge \sigma(\phi_1)$ we obtain (xiv) $\eta \models \phi'$. (xiii) and (xiv) imply $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$, which concludes the proof. □

**Lemma 2** (Precision)**.** *If $\pi \wedge^{\mathfrak{s}} \phi \stackrel{\alpha^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$ and $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$ then there exists a configuration $\gamma$ such that $\gamma \stackrel{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$.*

*Proof.* We assume again that $\alpha$ is $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$. From $\pi \wedge^{\mathfrak{s}} \phi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \pi' \wedge^{\mathfrak{s}} \phi'$ we obtain $\sigma : Var \cup \{\psi\} \to \mathcal{T}$ and a symbolic rule $\alpha^{\mathfrak{s}} \triangleq (\pi_1 \wedge^{\mathfrak{s}} \psi) \wedge sat(\psi \wedge \phi_1) \Rightarrow \pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1) \in \mathcal{S}^{\mathfrak{s}}$, obtained from $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \in \mathcal{S}$ such that (i) $\sigma(\pi_1 \wedge^{\mathfrak{s}} \psi) = \pi \wedge^{\mathfrak{s}} \phi$, (ii) $\sigma \models sat(\phi_1 \wedge \psi)$ and (iii) $\pi' \wedge^{\mathfrak{s}} \phi' = \sigma(\pi_2 \wedge^{\mathfrak{s}} (\psi \wedge \phi_1))$. From the hypothesis $\gamma' \in [\![\pi' \wedge^{\mathfrak{s}} \phi']\!]$ and (iii) we obtain a valuation $\rho$ such that (iv) $\gamma' = \rho(\pi') = \rho(\sigma(\pi_2))$ and (v) $\rho \models \sigma(\phi_1) \wedge \sigma(\psi)$. Let $\gamma \triangleq \rho(\sigma(\pi_1))$. We prove that $\gamma$ satisfies the lemma's conclusions.

$\gamma \Rightarrow_{\mathcal{S}} \gamma'$: we have (vi) $\gamma = \rho(\sigma(\pi_1)) = (\rho \circ \sigma)(\pi_1)$, from (iv) we obtain (vii) $\gamma' = (\rho \circ \sigma)(\pi_2)$, and from (v) we obtain (viii) $\rho \circ \sigma \models \phi_1$. (vi), (vii) and (viii) imply $\gamma \Rightarrow_{\mathcal{S}} \gamma'$.

$\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$: from (v) we also obtain (ix) $\rho \models \sigma(\psi)$ and using $\gamma \triangleq \rho(\sigma(\pi_1))$ we obtain (x) $\gamma \in [\![\sigma(\pi_1) \wedge^{\mathfrak{s}} \sigma(\psi)]\!]$. But $[\![\sigma(\pi_1) \wedge^{\mathfrak{s}} \sigma(\psi)]\!] = [\![\sigma(\pi_1 \wedge^{\mathfrak{s}} \psi)]\!] = [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$ - the latter equality is obtained using (i) - which proves $\gamma \in [\![\pi \wedge^{\mathfrak{s}} \phi]\!]$ and the lemma. □

This concludes the proof of Theorem 1. □

Proof of Theorem 2 (page 23). We prove first several lemmas, which state that the applications of equations and axioms may commute, and that $\rightarrow_{E/A}$ is ground terminating, ground confluent, and ground coherent.

**Lemma 3.** *Let $t_1, t_1', t_2$ be three ground terms such that $t_1 \rightarrow_e t_2$ and $t_1 =_{u=v} t_1'$ for some equation $e \in E$ and axiom $u = v \in A$. Then there exists $t_2'$ such that $t_1' \rightarrow_e t_2'$ and $t_2 =_{u=v} t_2'$.*

*Proof.* Assume that $e$ is $f(d_1, \ldots, d_n) = d$, where $d = \mathcal{D}_f(d_1, \ldots, d_n)$. Let $p, p'$ be two positions in $t_1$ and a ground substitution $\sigma$ such that $t_1|_p = f(d_1, \ldots, d_n)$, $t_2 = t_1[d]_p$ and $t_1|_{p'} = \sigma(u)$, $t_1' = t_1[\sigma(v)]_{p'}$ (or vice-versa, $t_1|_{p'} = \sigma(v)$, $t_1' = t_1[\sigma(u)]_{p'}$). We distinguish two cases:

1. $p$ and $p'$ are disjoint: then we take $t'_2 = t_1[d]_p[\sigma(v)]_{p'} = t_1[\sigma(v)]_{p'}[d]_p$.
2. Since $u = v$ is linear and regular and it includes only non-data function symbols, the only other possibility is that $f(d_1, \ldots, d_n)$ is a subterm of $\sigma(X)$ for some variable $X$. Let $\sigma'$ denote the substitution defined as follows: $\sigma'(X)$ is obtained by replacing the involved occurrence of $f(d_1, \ldots, d_n)$ by $d$ and $\sigma'(Y) = \sigma(Y)$ for $Y \neq X$. It is easy to see that $\sigma(X) \rightarrow_e \sigma'(X)$ and hence $\sigma(v) \rightarrow_e \sigma'(v)$ (that implies $t'_1 \rightarrow_e t'_1[\sigma'(v)]_{p'}$), and that $t_2|_{p'} = \sigma'(u)$. We take $t'_2 = t'_1[\sigma'(v)]_{p'}$.

So, in both cases we have $t_2 =_{u=v} t'_2$ and $t'_1 \rightarrow_e t'_2$. □

**Corollary 4.** *If $t_1 \rightarrow_E^* t_2$ and $t_1 =_A t'_1$ then there is $t'_2$ such that $t'_1 \rightarrow_E^* t'_2$ and $t_2 =_A t'_2$.*

**Lemma 4.** *$\rightarrow_{E/A}$ is ground terminating.*

*Proof.* Since the axioms $A$ are linear, regular, data collapse-free and apply only to non-data function symbols, it follows that if $t =_A t'$ then a data function symbol occurs in $t$ iff it occurs in $t'$ and hence $t$ and $t'$ have the same number of data function occurrences. The equations $f(d_1, \ldots, d_n) = \mathcal{D}_f(d_1, \ldots, d_n)$ may only be applied to ground $\Sigma$-terms, and each application strictly reduces the number of non-constant operations in $\Sigma^{\mathsf{Data}}$ in the term, by replacing non-constant terms $f(d_1, \ldots, d_n)$ by constants $\mathcal{D}_f(d_1, \ldots, d_n)$. So, $t \rightarrow_{E/A} t'$ implies that the number of occurrences of data function symbols in $t$ is greater than the number of occurrences of data function symbols in $t'$. Hence, there cannot be a nonterminating sequence of $\rightarrow_{E/A}$ reductions of a given ground term $t$, which proves the lemma. □

**Lemma 5.** *$\rightarrow_{E/A}$ is ground confluent.*

$$
\begin{array}{ccccccc}
t & =_A & t' & \rightarrow_{e'} & t_1 & =_A & t'_1 \\
=_A & & \text{Lemma 3} & & =_A & \text{Lemma 3} & \\
t'' & & \rightarrow_{e'} & & t''_1 & & e''\!\downarrow \\
e''\!\downarrow & & & & e''\!\downarrow & & \\
t_2 & & \rightarrow_{e'} & & t_3 & =_A & t''_3 \\
=_A & & \text{Lemma 3} & & =_A & & \\
t'_2 & & \rightarrow_{e'} & & t'_3 & &
\end{array}
$$

Figure .6: The proof of the confluence in Lemma 5

*Proof.* Since $\to_{E/A}$ is ground terminating, it is enough to show that it is ground *locally confluent* (and by Newman's lemma, it is also ground confluent). So we have to show that if $t \to_{e'/A} t'_1$ and $t \to_{e''/A} t'_2$, then there is $t_3$ such that $t'_i \to_{e'/A} t_3$, $i = 1, 2$. The situation is graphically represented by the diagram in Figure .6. Assume that $t \to_{e'/A} t_1$, thus, there are $t', t'_1$ such that $t =_A t', t' \to_{e'} t_1, t_1 =_A t'_1$. Similarly, if $t \to_{e''/A} t_2$ then there are $t'', t'_2$ such that $t =_A t'', t'' \to_{e''} t_2, t_2 =_A t'_2$. Then there is $t''_1$ such that $t_1 =_A t''_1, t'' \to_{e'} t''_1$ by Lemma 3.

We show now how $t_3$ is obtained. Assume that $e' \triangleq f'(d'_1, \ldots, d'_n) = d'$, $d' = \mathcal{D}_{f'}(d'_1, \ldots, d'_{n'})$, and $e'' \triangleq f''(d''_1, \ldots, d''_{n''}) = d''$, $d'' = \mathcal{D}_{f''}(d''_1, \ldots, d''_{n''})$, and that they match the term $t''$ at positions $p'$ and $p''$, respectively. Note that none of the positions may be a strict prefix of the other one, because if, say, $p''$ where a strict prefix of $p'$ then at last one among $d'_1, \ldots, d'_{n'}$ would not be a constant, in contradiction to the form of the equations $E$. Thus, we have the two following cases: If $p' = p''$ then $e'$ coincides with $e''$ and thus the confluence is not an issue. If $p'$ and $p'$ are disjoint positions, both $e'$ and $e''$ can be applied, which gives:

- $t''[d']_{p'}[d'']_{p''}$ if $e'$ is applied first, then $e''$ is applied;
- $t''[d'']_{p''}[d']_{p'}$ if $e''$ is applied first, then $e'$ is applied.

Since $p'$ and $p'$ are disjoint, the above terms are syntactically equal, say to $t_3$.

Finally, there are $t'_3$ and $t''_3$ such that $t'_2 \to_{e'} t'_3$, $t_3 =_A t'_3$ and $t'_1 \to_{e''} t''_3$, $t_3 =_A t''_3$, by applying twice Lemma 3. So, we may conclude that $t'_i \to_{E/A} t_3$, for $i = 1, 2$. □

**Lemma 6.** *$\to_{E/A}$ is ground coherent.*

$$\begin{array}{ccccccccccc}
t & =_A & t' & = & \sigma(\pi_1) & \to_\alpha & \sigma(\pi_2) & =_A & t_1 & \to^!_{E/A} & can_{E/A}(t_1) \\
{\scriptstyle E/A\downarrow_!} & & {\scriptstyle \text{confluence}} & & {\scriptstyle E/A\downarrow_!} & & {\scriptstyle \substack{\text{def.}\\ \text{of}\\ \sigma'}} & {\scriptstyle E/A\downarrow_*} & {\scriptstyle \text{confluence}} & & {\scriptstyle =_A} \\
can_{E/A}(t) & =_A & \sigma'(\pi_1) & & & \to_\alpha & \sigma'(\pi_2) & & & \to^!_{E/A} & can_{E/A}(\sigma'(\pi_2))
\end{array}$$

Figure .7: The proof of the coherence in Lemma 6

*Proof.* We have to show that for all $t, t_1 \in T_\Sigma$ with $t \to_{R/A} t_1$ there is $t_2 \in T_\Sigma$ s.t. $can_{E/A}(t) \to_{R/A} t_2$ and $can_{E/A}(t_1) =_A can_{E/A}(t_2)$. The situation

is graphically represented by the diagram in Figure .7. Assume that $t =_A t', t' \rightarrow_\alpha t'_2, t'_2 =_A t_1$, where $\alpha$ is a rule $\pi_1 \rightarrow \pi_2$ **if** $b$ in $R$. It follows that there is a ground substitution $\sigma$ such that $t' = \sigma(\pi_1)$, $t'_2 = \sigma(\pi_2)$ and $\vdash \sigma(b)$. Let $\sigma'$ denote the ground substitution defined by $\sigma'(X) = can_{E/A}(\sigma(X))$. It follows that $\sigma(\pi_1) \rightarrow^!_{E/A} \sigma'(\pi_1)$ (recall that any data subterm of $\pi_1$ is a variable) and $\sigma(\pi_2) \rightarrow^*_{E/A} \sigma'(\pi_2)$. We also have $\sigma'(\pi_1) \rightarrow_\alpha \sigma'(\pi_2)$ since $\vdash \sigma'(b)$ iff $\vdash \sigma(b)$ by the definition of $\vdash$ and that of $\sigma'$. Since $\rightarrow_{E/A}$ is ground confluent and terminating, it follows that $can_{E/A}(t) =_A \sigma'(\pi_1)$ and $can_{E/A}(t_1) =_A can_{E/A}(\sigma'(\pi_2))$. Now it is easy to see that $\sigma'(\pi_2)$ is just the term $t_2$ required by the coherence property. □

We come back now to the proof of Theorem 2. There exists a *matching algorithm modulo the equational axioms A* by Assumption 2.3. The termination, confluence and coherence properties follow from Lemma 4, Lemma 5, and Lemma 6, respectively. We show now that $\mathcal{T}_s = Can_{E/A,s}$., i.e., the statement of the theorem that identifies configurations (elements of $\mathcal{T}_{Cfg}$) and terms of sort $Can_{E/A,Cfg}$ is well formed.

We have $\mathcal{T}_s = \mathcal{D}\restriction_s^{\Sigma,\Pi}/=_A$ by Assumption 2. If $s$ is a data sort then $\mathcal{T}_s = \mathcal{D}_s = Can_{E/A,s}$ because each $d \in \mathcal{D}_s$ is a $E/A$-canonical form and each term $t \in T_{\Sigma^\mathfrak{d}}$ can be reduced using equations $E$ to an element $d \in \mathcal{D}_s$. Recall that a term of data sort may have only data subterms. If $f \in \Sigma \setminus \Sigma^\mathfrak{d}$ and $t_1, \ldots, t_n$ are $E/A$ canonical forms of appropriate sort, then $f(t_1, \ldots, t_n)$ is a $E/A$-canonical form, as well, because there are no equations $E$ involving $f$.

There remains to prove the faithfulness of the encoding (of language definitions into rewrite theories), i.e., $\gamma \stackrel{\alpha}{\Longrightarrow}_\mathcal{S} \gamma'$ iff $\gamma \stackrel{\alpha}{\rightarrow}_{\mathcal{R}(\mathcal{L})} \gamma'$. We have that $\gamma \stackrel{\alpha}{\Longrightarrow}_\mathcal{S} \gamma'$ iff $\alpha \triangleq (\pi_1 \wedge \phi_1 \Rightarrow \pi_2) \in \mathcal{S}$ and there is $\rho : Var \rightarrow \mathcal{T}$ are such that $\gamma =_\mathcal{T} \rho(\pi_1), \rho(\phi_1) =_\mathcal{T} true$ and $\gamma' = \rho(\pi_2)$.

Each valuation $\rho : Var \rightarrow \mathcal{T}$ defines a ground substitution $\sigma_\rho : Var \rightarrow T_\Sigma$ such that $\sigma_\rho(X) = \rho(X)$ for each $X \in Var$, which implies $can_{E/A}(\sigma_\rho(t)) = \rho(t)$ for each term $t \in \mathcal{T}_\Sigma(Var)$. For instance, with $A = \emptyset$, if $\rho(I_1) = 3, \rho(I_2) = 5$ then $\rho(I_1 +_{Int} I_2) = \mathcal{T}_{\_+_{Int}\_}(\rho(I_1), \rho(I_2)) = \mathcal{D}_{\_+_{Int}\_}(3, 5) = 8$ and $\sigma_\rho(I_1 +_{Int} I_2) = \_ +_{Int} \_(\sigma_\rho(I_1), \sigma_\rho(I_2)) = 3 +_{Int} 5$. The term $3 +_{Int} 5$ is reduced to 8 using the equations $E$.

Conversely, each ground substitution $\sigma : Var \rightarrow \mathcal{T}_\Sigma$ defines a valuation $\rho_\sigma : Var \rightarrow \mathcal{T}$ such that $\rho_\sigma(X) = can_{E/A}(\sigma(X))$ for each $X \in Var$.

If $\phi$ is a quantifier-free FOL formula (that can be represented as a term of sor $Bool_\mathcal{L}$ in $\mathcal{R}(\mathcal{L})$), then the claims 1) $\rho \models \phi$ iff $\vdash \rho_\sigma(\phi)$ and 2) $\vdash \sigma(\phi)$ iff $\rho_\sigma \models \phi$ are proved by structural induction on $\phi$ and using the definition of $\vdash$.

Thus, if a rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2$ in $\mathcal{S}$ is applicable on a concrete configuration $\gamma \in \mathcal{T}_{Cfg}$ via the valuation $\rho$ and produces $\gamma'$, then $\pi_1 \to \pi_2 \,\mathbf{if}\, \phi_1$ in $R$ is applicable to $\gamma$ via $\sigma_\rho$ and produces the same result $\gamma'$. Conversely, if a rule $\pi_1 \to \pi \,\mathbf{if}\, \phi_1$ in $R$ is applicable on $\gamma \in \mathcal{T}_{Cfg}$ via the substitution $\sigma$ and produces $\gamma'$, then $\pi_1 \wedge \phi_1 \Rightarrow \pi_2$ in $\mathcal{S}$ is applicable to $\gamma$ via $\rho_\sigma$ and produces the same result $\gamma'$. The faithfulness of the encoding is now proved, and the theorem is proved as well. □

Proof of Theorem 3 (page 24). We first introduce a definition and prove two lemmas.

**Definition 21.** A **data-abstraction** $t^\circ$ of a ground $\Sigma$-term $t$ w.r.t. a set $Y$ of variables (with $var(t) \subseteq Y$), and the substitution $\sigma_t^\circ$ associated to it, are defined as follows:

1. if $t$ is a data term, then $t^\circ \in Var \setminus Y$ is a fresh variable and $\sigma_t^\circ(t^\circ) = t$;
2. if $t = f(t_1, \ldots, t_n)$ with $f$ a non-data functional symbol, then $t^\circ = f(t_1^\circ, \ldots, t_n^\circ)$ and $\sigma_t^\circ = \sigma_{t_1}^\circ \uplus \ldots \uplus \sigma_{t_n}^\circ$, where $t_i^\circ$ is a data abstraction of $t_i$ w.r.t. $Y \cup \bigcup_{j \neq i} var(t_j^\circ)$ and $\sigma_{t_i}^\circ$ is the substitution associated to $t_i^\circ$.

*Remark* 3.
1. By Assumption 2.2, a term of data sort does not include non-data function symbols and hence any ground $\Sigma$-term has a data-abstraction that is unique up to a variable renaming.
2. A data-abstraction is linear since by construction any fresh variable occurs just once.
3. If $t^\circ$ is a data-abstraction of $t$ w.r.t. $Y$ and $\sigma_t^\circ$ is the substitution associated to $t^\circ$ then $\sigma_t^\circ(t^\circ) = t$.

All the above properties can be checked by induction on the number of non-data function symbols occurring in $t$.

**Lemma 7.** *Let $t$ be a linear term such that all its subterms of data sorts are variables, $\rho : Var \to \mathcal{T}$ a valuation, and $t^\circ$ a data-abstraction of $t$ w.r.t. $Y$. Then there is a substitution $\sigma : Var \to T_\Sigma(Var^{\mathfrak{d}})$ and a valuation $\eta : Var \to \mathcal{T}$ such that $\rho(y) = \eta(\sigma(y))$ for all $y \in Y$.*

*Proof.* We proceed by structural induction on $t$. We distinguish the following cases:

1. $t$ is a variable of data sort. We take $\sigma$ defined by:
   - $\sigma(t) = \rho(t)^\circ$ (which is a fresh variable because $\rho(t)$ is a data term)

- $\sigma(x) = x$ for all $x \neq t$

and the valuation $\eta$ given by:

- $\eta(\rho(t)^\circ) = \rho(t)$, and
- $\eta(x) = \rho(x)$ for all $x \neq \rho(t)^\circ$.

Thus, $\eta(\sigma(y)) = \rho(y)$ for all $y \in Y$ (note that $\rho(t)^\circ$ is not in $Y$).

2. $t$ is a variable of non-data sort.

   - $\sigma(t) = \rho(t)^\circ$
   - $\sigma(x) = x$ for all $x \neq t$

   and the valuation $\eta$ given by

   - $\eta(x) = \sigma^\circ_{\rho(t)}(x)$ for $x \in var(\rho(t)^\circ)$
   - $\eta(x) = \rho(x)$ for all $x \notin var(\rho(t)^\circ)$

   We have $\eta(\sigma(y)) = \rho(y)$ for all $y \in Y$ (note that $Y \cap var(\rho(t)^\circ) = \emptyset$).

3. $t = f(t_1, \ldots, t_n)$ where $f$ is a non-data function operation. Then $\rho(t)^\circ = f(\bar{t}_1, \ldots, \bar{t}_n)$ by Definition 21, where $\bar{t}_i = \rho(t_i)^\circ$ for $i = 1, \ldots n$. There are $\sigma_i$ and $\eta_i$ such that $\eta_i(\sigma_i(t_i)) = \rho(t_i)$ by the induction hypothesis ($t_i$ fulfils the hypotheses of the lemma). Let $\sigma$ denote the substitution defined by:

   - $\sigma(x) = \sigma_i(x)$ for all $x \in var(t_i)$ and $i = 1, \ldots, n$
   - $\sigma(x) = x$ for all $x \notin \bigcup_i var(t_i)$

   and let $\eta$ denote the valuation

   - $\eta(x) = \eta_i(x)$ for all $x \in var(\bar{t}_i)$ and $i = 1, \ldots, n$
   - $\eta(x) = \rho(x)$ for all $x \notin \bigcup_i var(\bar{t}_i)$

   Since $t$ is linear, it follows that $\sigma$ and $\eta$ are well-defined. We obtain $\rho(y) = \eta_i(\sigma_i(y))$ for all $y \in Y \cup \bigcup_{j \neq i} var(\bar{t}_j)$ and $i = 1, \ldots n$ by the induction hypothesis. Let $y \in Y$. We distinguish two cases: 1) $\sigma(y) = \sigma_i(y)$: then $y \in var(t_i)$ and $var(\sigma_i(y)) \subset var(\bar{t}_i)$, which implies $\eta(\sigma(y)) = \eta_i(\sigma_i(y)) = \rho(y)$; 2) $\sigma(y) = y$, which implies $\eta(y) = \rho(y)$ and hence $\eta(\sigma(y)) = \rho(y)$. So, in both cases we obtained $\eta(\sigma(y)) = \rho(y)$, which concludes the lemma.

□

**Corollary 5.** *In the context of Lemma 7:*

1. $\sigma(t) = \rho(t)^\circ$;
2. $\sigma(x) = \rho(x)^\circ$ *for all* $x \in var(t)$.

**Lemma 8.** *Let* $t$ *and* $t'$ *be two ground* $\Sigma$*-terms of the same non-data sort* $s$*. If* $t =_A t'$ *then there is a variable renaming* $\xi$ *such that* $t^\circ =_A \xi(t'^\circ)$.

*Proof.* The congruence $=_A$ on $T_\Sigma(Var)$ is the smallest equivalence relation that 1) includes $\rho(u) =_A \rho(v)$ for all $u = v$ in $A$ and $\rho : Var \to T_\Sigma(Var)$, and 2) is closed under replacement: if $t_1 =_A t_2$ then $t_0[t_1]_p =_A t_0[t_2]_p$.

We proceed by induction on the number of how many times the replacement rule is applied.

For the base case, we need to show that if $u = v$ in $A$, then $\rho(u)^\circ =_A \xi(\rho(v)^\circ)$ for some renaming $\xi$. Since $u = v$ is linear, regular, and data collapse-free we have $var(u) = var(v) = \{x_1, \ldots, x_n\}$. Moreover, since both $u$ and $v$ include only non-data functional symbols, we may apply Lemma 7 and Corollary 5 and obtain $\rho(u)^\circ = \sigma(u)$ and $\rho(v)^\circ = \sigma'(v)$, where $\sigma$ and $\sigma'$ are substitutions such that $\sigma(x_i)$ and $\sigma'(x_i)$ are equal to $\rho(x_i)^\circ$ up to a variable renaming $\xi_i$ (by the definition of data abstractions), $i = 1, \ldots, n$. The conclusion follows by taking $\xi = \xi_1 \uplus \ldots \uplus \xi_n$.

For the inductive setp: let $t = t_0[t_1]_p$ and $t' = t_0[t_2]_p$, $t_1 =_A t_2$. If $t_1$ and $t_2$ are of data sort then let $p'$ be the smallest nonstrict prefix of $p$ such that the subterm of $t_0$ at the position $p'$ is a data term. If it exists, the position $p'$ occurs in both $t^\circ$ and $t'^\circ$ and $t^\circ|_{p'}$, $t'^\circ|_{p'}$ are both fresh variables of data sort, thus, the substitution $\xi$ renaming $t'^\circ|_{p'}$ into $t^\circ|_{p'}$ ensures $t^\circ = \xi(t'^\circ)$, which solves this case.

If $p'$ does non exist, then $t_1$ and $t_2$ are of non-data sort. Then, we have $t_1^\circ =_A \xi(t_2^\circ)$, for some variable renaming $\xi$, by the induction hypothesis. Moreover the position $p$ occurs in both $t^\circ$ and $t'^\circ$ by Assumption 2.2.

So, we may assume w.l.o.g. that $t = t_0[t_1]_p$ and $t' = t_0[t_2]_p$, $t_1 =_A t_2$, $t_1^\circ =_A \xi(t_2^\circ)$ for some variable renaming $\xi$, and that the position $p$ occurs in both $t^\circ$ and $t'^\circ$. In this case we have $t^\circ = t_0^\circ[t_1^\circ]_p$ and $t'^\circ = t_0^\circ[t_2^\circ]_p$ (this can be checked by structural induction on $t_0$). From $t_1^\circ =_A \xi(t_2^\circ)$ we obtain $t_0^\circ[t_1^\circ]_p =_A t_0^\circ[\xi(t_2^\circ)]_p$ by the definition of $=_A$. The conclusion of the lemma follows by extending $\xi$ such that $t_0^\circ[\xi(t_2^\circ)]_p = \xi(t_0^\circ[t_2^\circ]_p)$. □

We now go back to the proof of Theorem 3. We may assume w.l.o.g. that $var(\pi_1) \cap var(\pi) = \emptyset$ and that the data-abstractions of $\rho(\pi)$ and $\rho(\pi_1)$ are computed w.r.t. $Y = var(\pi_1) \cup var(\pi)$, for any valuation $\rho$.

For the "if" implication: let $\sigma \in match_A(\pi_1, \pi)$, then, any valuation $\rho$ satisfying $\rho(\sigma(x)) = \rho(x)$ for all $x \in var(\pi_1)$ is a concrete unifier (we implicitly assume that all the carrier sets in $\mathcal{T}$ are non-empty, and use the fact that $var(\sigma(x)) \cap var(\pi_1) = \emptyset$ for all $x \in var(\pi_1)$, since $var(\sigma(x)) \subseteq Var^{\mathfrak{d}}$ for all $x \in var(\pi_1)$, and $var(\pi_1) \cap Var^{\mathfrak{d}} = \emptyset$.

For the "only if" part: We show that for any concrete unifier $\rho$ of $\pi_1$ and $\pi$ there is $\sigma \in match_A(\pi_1, \pi)$ (hence this set is not empty) and a valuation $\eta$ such that $\rho(x) = \eta(\sigma(x))$ for all $x \in var(\pi_1) \cup var(\pi)$. Recall that $\mathcal{T}^{\mathfrak{s}}_{Cfg} = T_{\Sigma(\mathcal{D}),Cfg}(Var^{\mathfrak{d}})$, $\mathcal{T} = \mathcal{D}\!\upharpoonright^{(\Sigma,\Pi)}\!/=_A$, and $\mathcal{D}\!\upharpoonright^{(\Sigma,\Pi)} = T_{(\Sigma\setminus\Sigma^{\mathfrak{d}})(\mathcal{D})}$. If $\pi_1$ and $\pi$ are concretely unifiable there is a valuation $\rho : Var \to T_{(\Sigma\setminus\Sigma^{\mathfrak{d}})(\mathcal{D})}$ such that $\rho(\pi_1) =_A \rho(\pi)$. We obtain $\rho(\pi_1)^\circ =_A \xi(\rho(\pi)^\circ)$ by Lemma 8, where $\xi$ is a variable renaming.

There are a substitution $\sigma_1$ and a valuation $\eta_1$ such that $\eta_1(\sigma_1(y)) = \rho(y)$, for all $y \in Y$, by Lemma 7. Moreover, $\sigma_1(\pi_1) = \rho(\pi_1)^\circ$ by Corollary 5.

Since $\pi$ includes only variables of data sort, it follows that any position $p$ in the data-abstraction $\rho(\pi)^\circ$ is a position in $\pi$ as well and hence $\pi$ is an instance of $\rho(\pi)^\circ$. It follows that $\pi$ is an instance of $\xi(\rho(\pi)^\circ)$ as well, i.e. there is a substitution $\sigma_2$ such that $\sigma_2((\xi(\rho(\pi)^\circ)) = \pi$. From $\rho(\pi_1)^\circ =_A \xi(\rho(\pi)^\circ)$ and $\sigma_1(\pi_1) = \rho(\pi_1)^\circ$ we obtain $\sigma_2(\sigma_1(\pi_1)) =_A \sigma_2(\xi(\rho(\pi)^\circ))$ ($=_A$ is closed under substitution), which implies $\sigma_2(\sigma_1(\pi_1)) =_A \pi$, i.e. $\sigma = \sigma_2 \circ \sigma_1$ is in $match_A(\pi_1, \pi)$.

Let $\eta$ denote the valuation defined by $\eta(y) = \rho(\sigma_2(y))$, for $y \in var(\xi(\rho(\pi)^\circ))$, and $\eta(x) = \eta_1(x)$ in the rest.

We prove that $\eta(\sigma(y)) = \rho(y)$ for all $y \in Y = var(\pi_1) \cup var(\pi)$. We distinguish the following two cases:

- $y \in var(\pi_1)$. From $\rho(\pi_1)^\circ =_A \xi(\rho(\pi)^\circ)$ and the linearity and regularity properties of the axioms $A$ we obtain $var(\rho(\pi_1)^\circ) = var(\xi(\rho(\pi)^\circ))$. Moreover, the abstraction mechanism ensures that $var(\rho(\pi_1)^\circ) \cap Y (= var(\pi_1) \cup var(\pi)) = \emptyset$, thus, $var(\rho(\pi_1)^\circ) \cap var(\pi_1) = \emptyset$. Then, $var(\pi_1) \cap var(\xi(\rho(\pi)^\circ)) = \emptyset$, and we have $\eta(\sigma(y)) = \eta_1(\sigma_1(y))$. We obtain $\eta_1(\sigma_1(y)) = \rho(y)$ by Lemma 7, hence $\eta(\sigma(y)) = \rho(y)$.

- $y \in var(\pi)$. Then $y \in Var^{\mathfrak{d}}$, which implies $\sigma_1(y) = y$. Then $\eta(\sigma(y)) = \eta(\sigma_2(y)) = \rho(y)$ by the definition of $\eta$.

We have proved $\sigma \in match_A(\pi_1, \pi) \neq \emptyset$ the existence of a valuation $\eta$ such that $\rho(x) = \eta(\sigma(x))$ for all $x \in var(\pi_1) \cup var(\pi)$. To complete the proof of the theorem we extend $\eta$ such that it coincides with $\rho$ on $Var \setminus (var(\pi_1) \cup var(\pi))$.

PROOF OF THEOREM 4 (PAGE 28). Let $\mathcal{R}(\mathcal{L})$ denote the rewrite theory $(\Sigma, E \cup A, \mathit{1R} \cup \mathit{2R}, \vdash)$. $\mathcal{R}(\mathcal{L})$ is built exactly as in Definition 17 by hypotheses. We prove a lemma first.

**Lemma 9.** *If $\gamma \to^*_{(E \cup \mathit{1R})/A} \gamma'$ 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 $\gamma \to^*_{(E \cup \mathit{1R})/A} \gamma' \triangleq \gamma_i$. When $i = 0$, then $\gamma' = \gamma$. Since all data subterms of $\gamma$ are constants, it follows that $\gamma$ is irreducible and we obviously have $\gamma \to^*_{\mathcal{R}(\mathcal{L})} \mathit{can}_{E/A}(\gamma) =_A \gamma$. We assume that $\gamma \to^*_{(E \cup \mathit{1R})/A} \gamma_i$ implies $\gamma \to^*_{\mathcal{R}(\mathcal{L})} \mathit{can}_{E/A}(\gamma_i)$ (♣), 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 $\gamma \to^*_{\mathcal{R}(\mathcal{L})} \mathit{can}_{E/A}(\gamma_{i+1})$, $i \geq 0$. There are two possibilities for $\gamma_{i+1}$ to be reached from $\gamma_i$:

- $\gamma_i \to_{E/A} \gamma_{i+1}$, i.e $\gamma_{i+1}$ is reached from $\gamma_i$ applying an equation. We obtain $\gamma_{i+1} \to^*_{E/A} \mathit{can}_{E/A}(\gamma_{i+1}) =_A \mathit{can}_{E/A}(\gamma_i)$ by the confluence and termination of $\to_{E/A}$ and $\gamma \to^*_{\mathcal{R}(\mathcal{L})} \mathit{can}_{E/A}(\gamma_{i+1})$ follows directly from (♣).

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

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

We now prove Theorem 4. Assume that $\gamma \twoheadrightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$. We have that $\varepsilon(\gamma) \to_{\mathit{2R}/A} \gamma''$, $\mathit{can}_{E/A}(\gamma'') =_A \gamma'$, and $\gamma \to^!_{(E \cup \mathit{1R})/A} \varepsilon(\gamma)$ by Definition 19. 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 $\gamma \to^*_{\mathfrak{R}(\mathcal{L})} \mathit{can}_{E/A}(\varepsilon(\gamma))$ (♢) by applying Lemma 9 for $\gamma \to^*_{(E \cup \mathit{1R})/A} \gamma_n$. Since $\varepsilon(\gamma)$ is $(E \cup \mathit{1R})$-irreducible, we have $\mathit{can}_{E/A}(\varepsilon(\gamma)) =_A \varepsilon(\gamma)$ and hence (♢) is equivalent to $\gamma \to^*_{\mathfrak{R}(\mathcal{L})} \varepsilon(\gamma)$. From $\varepsilon(\gamma) \to_{\mathit{2R}/A} \gamma''$ and $\mathit{can}_{E/A}(\gamma'') =_A \gamma'$ we have that $\varepsilon(\gamma) \to_{\mathfrak{R}(\mathcal{L})} \gamma'$. We obtain $\gamma \Rightarrow^*_{\mathcal{S}} \varepsilon(\gamma) \Rightarrow_{\mathcal{S}} \gamma'$ by applying Theorem 2 to $\gamma \to^*_{\mathfrak{R}(\mathcal{L})} \varepsilon(\gamma)$ and $\varepsilon(\gamma) \to_{\mathfrak{R}(\mathcal{L})} \gamma'$, respectively. Hence we conclude that $\gamma \Rightarrow^+_{\mathcal{S}} \gamma'$.

PROOF OF THEOREM 5 (PAGE 29). Assume $\gamma \rightarrow_{\mathfrak{R}(\mathcal{L})} \gamma'$. By the proof of Theorem 4 and Theorem 2 we have $\gamma = \gamma_0 \stackrel{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \stackrel{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \stackrel{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n = \gamma'$. 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 $\pi \wedge^{\mathfrak{s}} \phi = \pi_0 \wedge^{\mathfrak{s}} \phi_0 \stackrel{\alpha_1^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge^{\mathfrak{s}} \phi_1 \stackrel{\alpha_2^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \stackrel{\alpha_n^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge^{\mathfrak{s}} \phi_n$ such that $\gamma_i \models \pi_i \wedge^{\mathfrak{s}} \phi_i$. 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 $\pi_{n-1} \wedge^{\mathfrak{s}} \phi$ is $\mathit{1R}^{\mathfrak{s}}$-irreducible modulo $A$ (otherwise $\gamma_{n-1}$ 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, $\pi_{n-1} \wedge^{\mathfrak{s}} \phi = \varepsilon^{\mathfrak{s}}(\pi \wedge^{\mathfrak{s}} \phi)$ and hence $\pi \wedge^{\mathfrak{s}} \phi \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi_n \wedge^{\mathfrak{s}} \phi_n$.

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

□

PROOF OF THEOREM 6 (PAGE 29). Assume $\pi \wedge^{\mathfrak{s}} \phi \rightarrow_{\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})} \pi' \wedge^{\mathfrak{s}} \phi'$. By the proof of Theorem 4 and Theorem 2 we have $\pi \wedge^{\mathfrak{s}} \phi = \pi_0 \wedge^{\mathfrak{s}} \phi_0 \stackrel{\alpha_1^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_1 \wedge^{\mathfrak{s}} \phi_1 \stackrel{\alpha_2^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \cdots \stackrel{\alpha_n^{\mathfrak{s}}}{\Longrightarrow}_{\mathcal{S}^{\mathfrak{s}}} \pi_n \wedge^{\mathfrak{s}} \phi_n = \pi' \wedge^{\mathfrak{s}} \phi'$, 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 $\gamma_0 \stackrel{\alpha_1}{\Longrightarrow}_{\mathcal{S}} \gamma_1 \stackrel{\alpha_2}{\Longrightarrow}_{\mathcal{S}} \cdots \stackrel{\alpha_n}{\Longrightarrow}_{\mathcal{S}} \gamma_n = \gamma'$, such that $\gamma_i \models \pi_i \wedge^{\mathfrak{s}} \phi_i$. 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 $\gamma_0 \models \pi \wedge^{\mathfrak{s}} \phi$. Since the strategies $\varepsilon^{\mathfrak{s}}$ and $\varepsilon$ are isomorphic, it follows that $\gamma_{n-1} = \varepsilon(\gamma)$ because $\pi_{n-1} \wedge^{\mathfrak{s}} \phi_{n-1} = \varepsilon^{\mathfrak{s}}(\pi_0 \wedge^{\mathfrak{s}} \phi_0)$. □