WRLA'12, in Selected papers LNCS, to appear
LNCS 7571, pp 31-53. 2012

𝕂 Framework Distilled*

Dorel Lucanu[1], Traian Florin Șerbănuță[1,2], and Grigore Roșu[1,2]

[1] Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania, `dlucanu@info.uaic.ro`
[2] Department of Computer Science
University of Illinois at Urbana-Champaign, USA, `grosu@illinois.edu`

Abstract. 𝕂 is a rewrite-based executable semantic framework in which programming languages, type systems, and formal analysis tools can be defined using configurations, computations and rules. Configurations organize the state in units called cells, which are labeled and can be nested. Computations are special nested list structures sequentializing computational tasks, such as fragments of program. 𝕂 (rewrite) rules make it explicit which parts of the term they read-only, write-only, read-write, or do not care about. This makes 𝕂 suitable for defining truly concurrent languages even in the presence of sharing. Computations are like any other terms in a rewriting environment: they can be matched, moved from one place to another, modified, or deleted. This makes 𝕂 suitable for defining control-intensive features such as abrupt termination, exceptions or call/cc. This paper presents an overview of 𝕂 Framework and the 𝕂 tool, focusing on the interaction between the 𝕂 tool and Maude.

1 Introduction

Introduced by the second author in 2003 for teaching programming languages [1], and continuously refined and developed ever since (see, e.g., [2,3]), 𝕂 is a programming language definitional framework which aims to bring together the collective strengths of existing frameworks (expressiveness, modularity, concurrency, and simplicity) while avoiding their weaknesses. The 𝕂 framework has already been used to define real-life programming languages, such as C, Java, Scheme, and several program analysis tools (see Section 7 for references). 𝕂 is representable in rewriting logic, and this representation has been automated in the 𝕂 tool for execution, testing and analysis purposes using Maude [4].

This paper gives a brief overview of the 𝕂 framework and its current implementation, focusing on: (1) its place in the rewriting logic semantics [5,6,7] project; (2) its main features; (3) how easy it is to define programming language features in 𝕂; (4) how to use the 𝕂 tool to compile 𝕂 definitions into Maude, and to execute and analyze programs against these definitions.

The remainder of this paper is organized as follows. Section 2 motivates 𝕂 and places it in the general programming language semantics research context, and in particular within the rewriting logic semantics project. Section 3 gives

* This work is supported by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK).

an overview of the main features of $\mathbb{K}$. Section 4 shows $\mathbb{K}$ at work, by giving a compact semantics for a combination of the call-by-value and call-by-reference parameter passing styles. Section 5 discusses the transition semantics associated to a $\mathbb{K}$ definition and its relation to our current embedding of $\mathbb{K}$ into rewriting logic. Section 6 shows how the embedding of $\mathbb{K}$ into Maude through the $\mathbb{K}$ tool can be used to execute, explore, and model check programs. Section 7 concludes.

The didactic language CinK [8] is used as a running example.

2 Rewriting Logic Semantics, Related Work, Motivation

The research presented in this paper is part of the rewriting logic semantics project [5,6,7], an international collaborative effort to advance the use of rewriting logic for defining programming languages and for analyzing programs.

Rewriting is an intuitive and simple mathematical paradigm which specifies the evolution of a system by matching and replacing parts of the system state according to *rewrite rules*. Besides being formal, rewriting is also executable, by simply repeating the process of rewriting the state. Additionally, an initial state together with a set of rules yields not only a formal execution of the system, but also a transition system comprising all the system behaviors, which can be thus formally analyzed. Moreover, a rewriting semantic definition of a language can also be used to (semi-)automate the verification of programs written in that language, by using the semantic rules to perform symbolic execution of the program and hereby discharging (some of) the proof obligations.

Rewriting logic [9] combines term rewriting and equational logic in a formalism suitable to define truly concurrent systems. Equations typically define structural identities between states, and rewrite rules apply modulo equational rearrangements of the state. The benefits of using rewriting logic in defining the behavior of systems are multiple. First, one directly gains executability, and thus the ability to directly use formal definitions as interpreters. Second, it allows to capture the intended concurrency of the defined system directly in the definition, rather than relying on subsequent abstractions. Furthermore, by encoding the deterministic rules of a rewrite system as equations [10], the state-space of the resulting transition systems is drastically reduced, thus making its exploration more feasible and practical. The Maude rewrite system [4] offers a suite of tools for rewrite theories: debugger, execution tracer, state-space explorer, explicit-state LTL model checker, inductive theorem prover, etc. For example, model checking Java programs in Maude using a definition of Java, following the $\mathbb{K}$ technique presented here, was shown to compare favorably [11] with Java PathFinder, the state-of-art explicit-state model checker for Java [12].

When defining a language semantics in rewriting logic, the program state is typically represented as a configuration term. Equations represent structural rearrangements of the configuration or behaviorally irrelevant computational steps. Rewrite rules capture the relevant computational steps, namely those which we want to count as actual transitions between states. This way, a program execu-

WRLA'12, in LNCS, to appear

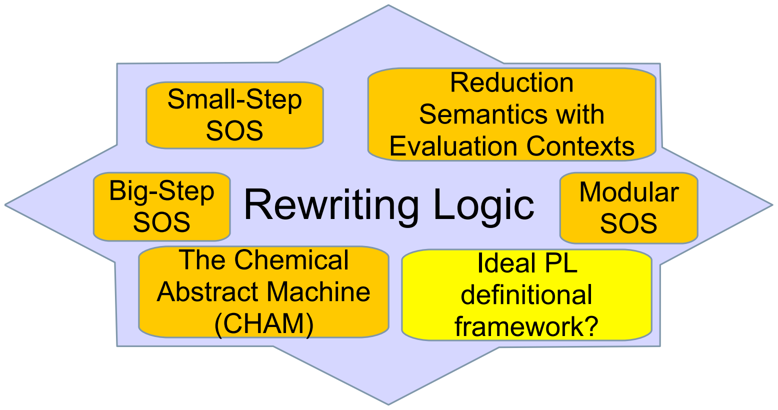

Small-Step SOS
Reduction Semantics with Evaluation Contexts
Big-Step SOS
Rewriting Logic
Modular SOS
The Chemical Abstract Machine (CHAM)
Ideal PL definitional framework?

Fig. 1. Rewriting logic as a meta-logical framework for defining programming languages

tion is captured as a sequence of transitions between equivalence classes of configuration terms, and the state-space of executions is captured as the transition system defined by the rewrite rules. Several paradigmatic languages have been given a faithful rewriting logic semantics this way [9,13,14], and even some small programming languages, following different styles and methodologies. In fact, [15] shows how various operational semantics approaches can be framed as methodological fragments of rewriting logic, including big-step (or natural) semantics [16], (small-step) structural operational semantics (SOS) [17], Modular SOS (MSOS) [18], reduction semantics (with evaluation contexts) [19], continuation-based semantics [20], and the chemical abstract machine (CHAM) [21].

Thus, we can regard rewriting logic as a meta-framework for defining programming language semantics, as illustrated in Figure 1. Once a language is defined in rewriting logic, the arsenal of generic tools of the latter can then be used to formally analyze both the programming language itself as well as its programs. An advantage that rewriting logic offers over other similar powerful meta-frameworks, such as e.g., higher-order logic, is that it allows us to tune the computational granularity of the defined language, both in depth (when we want several small steps to count as one step) and in breadth (when we want several non-overlapping steps to proceed concurrency), with little or no effort.

Unfortunately, one pragmatic problem that all meta-frameworks share is that they do not tell us *how* to define a language. They only give us powerful means to faithfully and uniformly represent any semantics, following any approach, using the same formalism. This desirable faithfulness actually also implies that a meta-framework, no matter how powerful it is, cannot magically eliminate the inherent limitations of the chosen semantic approach. For example, existing approaches have problems with control-intensive features (except for evaluation contexts), with modularity (except for MSOS, and except for evaluation contexts in some cases), with true concurrency (except for the CHAM), and so on.

Ideally, we would like a semantic approach, or framework, which has at least the union of all the strengths of the existing approaches, and which at the same time avoids all their weaknesses. Such a framework would also likely be rep-

WRLA'12, invited paper, to appear
LNCS 7571, pp 31-53, 2012

resentable in powerful meta-frameworks such as rewriting logic or higher-order logic, but that is not the point here. The point is that it is not clear whether such a framework is possible. In particular, such a framework should be at least as expressive as reduction semantics with evaluation contexts, but should also allow non-syntactic, environment/store style definitions; it should be at least as modular as MSOS, but should also give us access to the execution/evaluation context; it should be at least as concurrent as the CHAM, but should not force us artificially encode everything in molecules and solutions; and so on. Whether $\mathbb{K}$ has all these desirable features is and probably will always be open for debate. Nevertheless, $\mathbb{K}$ has been from the very beginning designed in a bottom-up fashion, striving to incorporate the positive aspects of the existing approaches and to avoid their negative aspects, at the same time being based on a rigorous mathematical foundation and offering an intuitive notation to its users.

3 The $\mathbb{K}$ Framework

In a nutshell, the $\mathbb{K}$ framework consists of computations, configurations, and rules. *Computations* are special sequences of tasks, where a task can be, e.g., a fragment of program that needs to be processed. *Configurations* are organized as nested soups of cells that hold syntactic and semantic information. $\mathbb{K}$ *rules* distinguish themselves by specifying only what is needed from a configuration, and by clearly identifying what changes, and thus, being more concise, more modular, and more concurrent than regular rewrite rules.

The running example of this paper is CinK [8], an overly-simplified kernel of the C++ language including integer and boolean expressions, functions, and basic imperative statements. Without modifying anything but the configuration, the language is extended with the following concurrency constructs: thread creation, lock-based synchronization and thread join.

Configurations. The initial running configuration of CinK is presented in Figure 2. The configuration is a nested multiset of labeled cells, in which each cell can contain either a list, a set, a bag, a map, or a computation. The initial CinK configuration consists of a top cell, labeled "T", holding a bag of cells, among which a map cell, labeled "store", to map locations to values, a list cell, labeled "in", to hold input values, and a bag cell, labeled "threads", which can hold any number of "thread" cells (signaled by the star "$*$" attached to the label of the cell). The thread cell is itself a bag of cells, among which the "k" cell holds a computation structure, which plays the role of directing the execution.

Syntax and Computations. Computations extend the user-defined language syntax with a task sequentialization operation, "$\curvearrowright$". The basic unit of computation is a task, which can be either a fragment of syntax, possibly with holes in it, or a semantic task, such as an environment recovery. Most of the manipulation of the computation is abstracted away from the language designer via intuitive PL syntax annotations like strictness constraints which, when declaring the syntax of a construct also specify the order of evaluation for its arguments. Similar decompositions of computations happen in abstract machines by means of stacks [20],

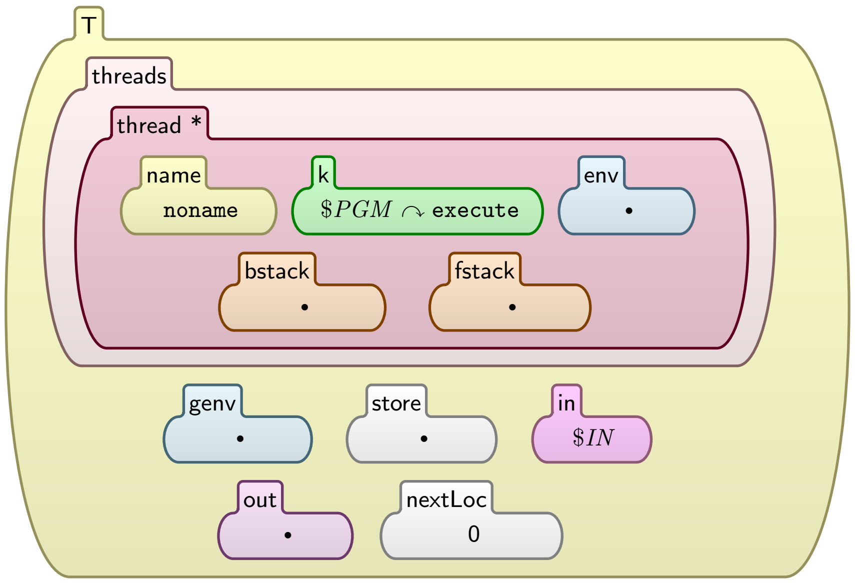


Fig. 2. The initial configuration of the CinK language.

and also in the refocusing techniques for implementing reduction semantics with evaluation contexts [22]. However, what is different here is that $\mathbb{K}$ achieves the same thing *formally*, by means of rules (there are special rules behind the strictness annotations, as explained below), not as an implementation means.

The $\mathbb{K}$ BNF syntax specified below suffices to parse the program fragment "`t = * x; * x = * y; * y = t;`" specifying a sequence of statements for swapping the values at two memory locations:

$$\begin{array}{lrl} \textsc{syntax} & \mathit{Exp} ::= & \mathit{Id} \\ & \mid & {*}\ \mathit{Exp}\ [\mathrm{strict}] \\ & \mid & \mathit{Exp} = \mathit{Exp}\ [\mathrm{strict}(2)] \end{array}$$

$$\begin{array}{lrl} \textsc{syntax} & \mathit{Stmt} ::= & \mathit{Exp}\ ;\ [\mathrm{strict}] \\ & \mid & \mathit{Stmt}\ \ \mathit{Stmt}\ [\mathrm{seqstrict}] \end{array}$$

Strictness annotations add semantic information to syntax by specifying the order of evaluation of arguments. The special rules corresponding to these strictness annotations are a special case of *structural rules*, metaphorically called heating/cooling rules like in the CHAM with the one going from left to right called a heating rule and the one from right to left called a cooling rule, are:

$$\begin{array}{c} {*}\ \mathit{ERed} \rightleftharpoons \mathit{ERed} \curvearrowright {*}\,\square \\ E = \mathit{ERed} \rightleftharpoons \mathit{ERed} \curvearrowright E = \square \\ \mathit{ERed}\ ; \rightleftharpoons \mathit{ERed} \curvearrowright \square\ ; \\ \mathit{SRed}\ \ S \rightleftharpoons \mathit{SRed} \curvearrowright \square\ \ S \\ \mathit{Val}\ \ \mathit{SRed} \rightleftharpoons \mathit{SRed} \curvearrowright \mathit{Val}\ \ \square \end{array}$$

The heating/cooling rules specify that the arguments mentioned in the strictness constraint can be taken out for evaluation at any time and plugged back into their original context. Note that statement composition generates two such rules (as, by default, strictness applies to each argument); however, since the constraint specifies *sequential strictness*, the second statement can be evaluated only once the first statement was completely evaluated (specified by the *Val* variable which should match a value) and its side effects were propagated.

By successively applying the heating/cooling rules above on the statement sequence above, we obtain the following (structurally equivalent) computations:

$$\begin{array}{r}
\texttt{t = * x; * x = * y; * y = t;} \rightleftharpoons \\
\texttt{t = * x;} \curvearrowright \square\ \texttt{* x = * y; * y = t;} \rightleftharpoons \\
\texttt{t = * x} \curvearrowright \square\texttt{;} \curvearrowright \square\ \texttt{* x = * y; * y = t;} \rightleftharpoons \\
\texttt{* x} \curvearrowright \texttt{t =} \square \curvearrowright \square\texttt{;} \curvearrowright \square\ \texttt{* x = * y; * y = t;} \rightleftharpoons \\
\texttt{x} \curvearrowright \texttt{*} \square \curvearrowright \texttt{t =} \square \curvearrowright \square\texttt{;} \curvearrowright \square\ \texttt{* x = * y; * y = t;}
\end{array}$$

The heating rules thus pull redexes out from their context for evaluation according to the desired evaluation strategy of the corresponding constructs, leaving holes as placemarkers for where to plug their results or intermediate computations back using the cooling rules. Above, the heating rules eventually singled out the variable `x` at the top of the computation. As seen shortly, other rules can now match it and replace it with its corresponding value from the store. The cooling rules can then plug that value back into its place in context.

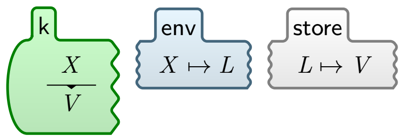
Implementations can choose to keep computations heated as an optimization, and only cool by need and only as much as necessary. Nevertheless, from a theoretical perspective, heating/cooling rules can be applied at any time and as many times as they match, thus yielding a potentially exponential number of structurally equivalent computations. As seen in Section 6, these can lead to non-deterministic behaviors of programs.

$\mathbb{K}$ *rules.* As discussed above, $\mathbb{K}$ has a particular kind of rules, called structural rules, which allow us to rearrange the configuration. Heating/cooling rules are a special kind of structural rules, which are typically bidirectional. $\mathbb{K}$ also allows standalone structural rules, for example a rule desugaring a “for” loop into a “while”, which need not be reversible. The distinction between heating/cooling rules and other structural rules is purely methodological, with no semantic implications. Because of that, one should feel free to call other pairs of structural rules, which do not necessarily capture evaluation strategies, also heating/cooling. For example, pairs of structural rules corresponding to intended equations (“heat” $A*(B+C)$ into $A*B+A*C$, and “cool” $A*B+A*C$ into $A*(B+C)$).

In addition to structural rules, $\mathbb{K}$ also has *computational rules.* The distinction between structural and computational rules is purely semantic, and will be clarified in Section 5. Intuitively, only the computational rules yield transitions in the transition system associated to a program. The role of the structural rules is to only rearrange the configuration so that computational rules can match.

The computational rule below succinctly describes the intuitive semantics for reading the value of a variable: if variable X is the next thing to be evaluated and if X is mapped to a location L in the environment, and that location is mapped to a value V in the store, then replace that occurrence of X by V. Moreover, note that the rule only specifies what is needed from the configuration, which is essential for obtaining modular definitions, and by precisely identifying what changes, which significantly enhances modularity and concurrency.

RULE MEM-LOOKUP-R-VALUE



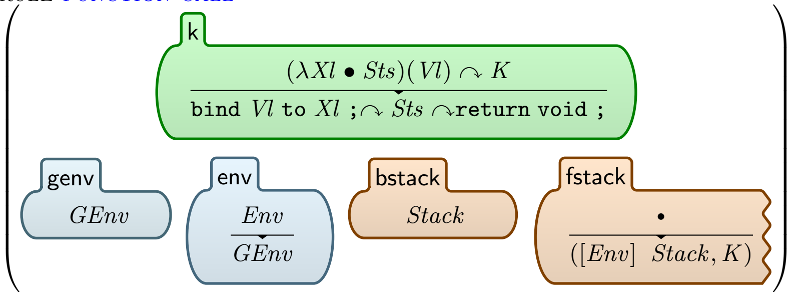
There are several ways in which $\mathbb{K}$ rules differ from regular rewrite rules. First, *in-place rewriting* allows one to specify small changes into a bigger context, by underlining the part that needs to change and writing its replacement under the line, instead of repeating the context in both sides of a rewrite rule. This additionally gives us the ability of using anonymous variables for the unused variables in the context, and, furthermore, the use of *cell comprehension* for focusing only on the parts of the cells which are relevant for this rule. Our metaphorical notation for cell comprehension is the jagged cell edge, which thus specifies that there could be more items in the cell, in the corresponding side, in addition to what is explicitly specified. Finally, the process of *configuration abstraction* allows for only the relevant cells to be mentioned in a rule, relying on the static structure of the declared configuration to infer the rest.

Modularity. Configuration abstraction is crucial for modularity. Relying on the initial configuration to be specified by the designer, and on the fact that usually the structure of such a configuration does not change during the execution of a program, the $\mathbb{K}$ rules are essentially invariant under change of configuration structure. This effectively means that the same rule can be re-used in different definitions as long as the required cells are present, regardless of the additional context, which can be automatically inferred from the initial configuration.

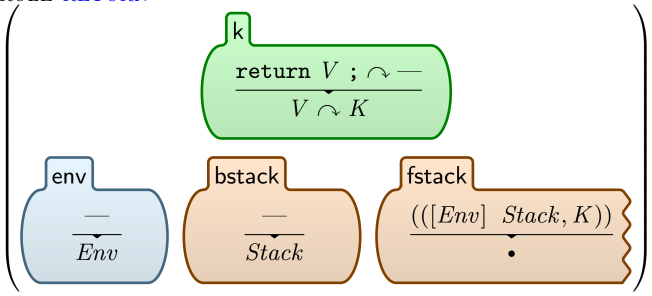
Expressiveness. The particular structure of $\mathbb{K}$ computations, and the fact that the current task is always at the top of the computation, greatly enhances the expressiveness of the $\mathbb{K}$ framework. Next paragraphs show how easy it is to use $\mathbb{K}$ to define constructs which are known to be hard to define in other frameworks.

Abrupt returning from a function is hard to define in many frameworks (except for reduction semantics with evaluation contexts), due to the lack of explicit access to the execution context. Having the entire remainder of computation always following the current redex allows the $\mathbb{K}$ definition of CinK to capture this construct in a simple and succinct manner by the following two rules:

RULE FUNCTION-CALL



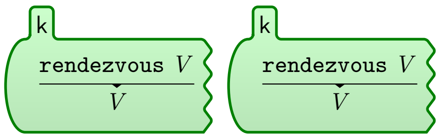
RULE RETURN



The function name is evaluated to its value, which is a lambda abstraction: *Xl* is the list of parameters, *Sts* is body of the function. The FUNCTION-CALL rule pushes the calling context, i.e., the remainder of the computation K and environment stack (including the current environment) on top of the function stack, while the RETURN rule uses the information there to restore the environment and computation of the caller. The evaluation of the arguments *Vl* and their binding to the formal parameters is described in Section 4.

Another feature which is hard to represent in other frameworks is handling multiple tasks at the same time, as when defining synchronous communication, for example. Although SOS-based frameworks can capture specific versions of this feature for languages like CCS or the π-calculus, they can only do it there because the communication commands are always at the top of their processes. $\mathbb{K}$ computation's structure is again instrumental here, as it allows to easily match two redexes at the same time, as shown by the following rule, defining the semantics of a `rendezvous` expression used for synchronizing two threads:

RULE RENDEZVOUS



Reading this rule one can easily get the intended semantics: a thread requesting a `rendezvous` has to wait until another thread makes a request with the same value V; once that happens, both threads can continue with V as their result.

$\mathbb{K}$ has a reflective view of syntax. Although it allows us to use concrete syntax in definitions as a convenience, it regards all syntactic terms as abstract syntax

WRLA'12, LNCS 7571, pp 31-53, 2012
WRLA'12, invited paper, to appear

trees (AST). Thus, language constructs are regarded as AST labels. For example, $a + 3$ is represented in $\mathbb{K}$ as $\_+\_(a(\bullet_{List\{K\}}), 3(\bullet_{List\{K\}}))$. This abstract view of syntax allows reducing the computation constructs to the following core:

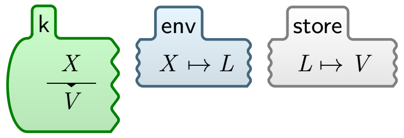
$$\begin{array}{lrl}
\textsc{syntax} & K ::= & KLabel(List\{K\}) \\
 & \mid & \bullet_K \\
 & \mid & K \curvearrowright K \\
\textsc{syntax} & List\{K\} ::= & K \\
 & \mid & \bullet_{List\{K\}} \\
 & \mid & List\{K\}, List\{K\}
\end{array}$$

We won't go into details here, but the ability of referring to the $\mathbb{K}$ AST in a definition allows $\mathbb{K}$ to define powerful reflective rules for AST manipulation such as generic AST visitor patterns, code generation, or generic substitution [23].

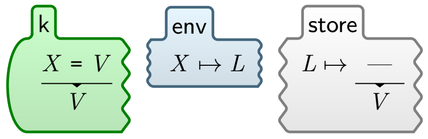
Concurrency. An aspect that makes $\mathbb{K}$ appropriate for defining programming languages is its natural way to capture concurrency. Besides being truly concurrent (like CHAM), $\mathbb{K}$ also allows capturing concurrency with resource sharing.

Let us exemplify this concurrency power. The two rules below specify the semantics for accessing/updating the value at a memory location:

RULE MEM-LOOKUP-R-VALUE



RULE ASSIGNMENT



As the semantics of the $\mathbb{K}$ rules specify that the parts of the configuration which are only read by the rule can be shared by concurrent applications, the read rule can match simultaneously for two threads attempting to read from the same location, and they can both advance one step concurrently. A similar thing happens for concurrent updates. As long as the threads attempt to update distinct locations, the update rules can match at the same time and the threads can advance concurrently. Moreover, by disallowing rule instances to overlap on the parts they change, the $\mathbb{K}$ semantics enforces sequentialization of data-races.

4 Case Study: Parameter Passing Styles in CinK

In this section we exhibit the definitional power of $\mathbb{K}$, by using it to compactly and naturally define a non-trivial language feature, namely the combination between call-by-value and call-by-reference as mechanisms for binding the formal parameters of a function to the arguments passed during a function call.

For call-by-value, the arguments passed to a function call are first evaluated in the context of the caller, then their values are stored into fresh memory locations, which are then bound to the corresponding formal parameters of the function. The lifetime of these fresh memory locations is limited to the execution of the called function's body, which guarantees that they are not accessible by the caller function after the callee's return.

For call-by-reference, the arguments must evaluate to l-values, and the formal parameters are directly bound to the locations designated by the resulting l-values. Therefore, any updates to the formal parameters during the execution of the function body is reflected onto the arguments passed to the call.

In this section we show how the two mechanisms are being combined in the $\mathbb{K}$ definition of CinK. CinK uses a C++-like notation for the two mechanisms. For instance, the code below declares a function `f` with two parameters `x` and `y`, `x` being called-by-value, while `y` being called-by-reference:

```
int f(int x, int &y) {
  y = ++x;
  return x;
}
```

L-value and R-value Expressions. CinK expressions can be evaluated to either *l-values* or *r-values*. Historically, the names of these two categories come from the fact that an l-value can be used in the left hand side of an assignment, i.e., it can be assigned to, while an r-value corresponds to the right hand side of an assignment (it can be assigned). Semantically, an l-value expression is evaluated to a location, while an r-value expression is evaluated to a value that can be stored into a location. Locations in our $\mathbb{K}$ definition of CinK are modeled by non-negative integers; we write `loc` (L) whenever the value L designates a location. This is achieved with the following syntax declaration:

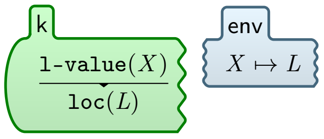
SYNTAX $\quad Val ::= \texttt{loc}(Int)$

To distinguish expressions which must evaluate to l-values we introduce a special wrapper for them (the other expressions are considered r-values by default):

SYNTAX $\quad Exp ::= \texttt{l-value}(K)$

The rule evaluating a program variable to an r-value is given on page 9. It replaces a variable (at the top of the computation) with the value stored in the location associated to that variable. In contrast, the rule that evaluates a program variable to its l-value replaces the variable with the location associated to it:

RULE MEM-LOOKUP-L-VALUE



Function call. The rule defining the evaluation of a function call expression is described on page 8. It assumes that the arguments have already been evaluated, binds their values to the formal parameters, and schedules the body for execution, while saving the calling context to be restored when returning from the call.

This rule is rather plain, similar to languages with a single evaluation strategy; therefore, it does not explain how the actual parameters are evaluated. If the language included only the call-by-value mechanism, then it would be enough to declare the function call expression strict in both arguments. However, since the evaluation strategy for the second argument is depending on the binding specification in the function signature, the function call expression is declared strict only in its first argument:

SYNTAX $\quad Exp ::= Exp\ \texttt{(}\ Exps\ \texttt{)}\ [\text{strict}(1)]$

and a more complex mechanism for evaluating the parameters is required.

Parameter passing styles. To evaluate the arguments of a function call according to the strategy specified by the function parameters, we use 𝕂's special support for evaluation contexts. A context declaration for the function call specifies that the evaluation of arguments needs to consider their declared strategy:

CONTEXT: $(\lambda Xl \bullet Sts)(\underbrace{\qquad\qquad\square\qquad\qquad}_{\texttt{evaluate}\ \square\ \texttt{following}\ Xl\ \texttt{;}})$

This context says not only that the actual parameters must be evaluated when passed to a function value, but also that they need to be evaluated using the `evaluate` construct and `following` the list of formal parameters.

SYNTAX $\quad Exps ::= \texttt{evaluate}\ Exps\ \texttt{following}\ Decls\ \texttt{;}$

For a call-by-value formal parameter, the corresponding argument must be evaluated normally, to an r-value:

CONTEXT: $\texttt{evaluate}\ \square\texttt{,}\ \text{—}\ \texttt{following}\ \texttt{int}\ X\ \texttt{,}\ \text{—}\ \texttt{;}$

For a call-by-reference formal parameter, the corresponding argument must be evaluated as an l-value expression:

CONTEXT: $\texttt{evaluate}\ \underbrace{\qquad\square\qquad}_{\texttt{l-value}\ (\square)}\texttt{,}\ \text{—}\ \texttt{following}\ \texttt{int}\ \texttt{\&}\ X\ \texttt{,}\ \text{—}\ \texttt{;}$

This second context uses again the special type of context used above for `evaluate`, by requesting that the expression on position $\square$ be evaluated as an l-value.

The following two rules, together with the strict evaluation strategy for lists of expressions complete the semantics of `evaluate` by recursing into the lists:

RULE
$\underbrace{\texttt{evaluate}\ V\ \texttt{,}\ El\ \texttt{following}\ \text{—}\texttt{,}\ Xl\ \texttt{;}}_{V\ \texttt{,}\ \texttt{evaluate}\ El\ \texttt{following}\ Xl\ \texttt{;}}$

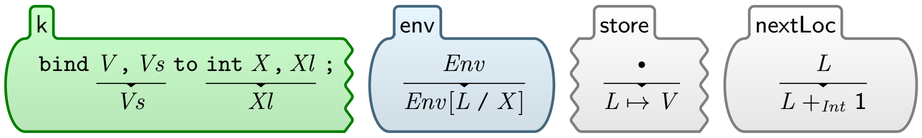
RULE
$\underbrace{\texttt{evaluate}\ \bullet\ \texttt{following}\ \bullet\ \texttt{;}}_{\bullet}$

Binding mechanisms. Similarly to the evaluation rules, the binding rules are also different for the two parameter passing styles. The binding is performed using an auxiliary construction:

SYNTAX $\quad K ::= \texttt{bind}\ Vals\ \texttt{to}\ Decls\ \texttt{;}$

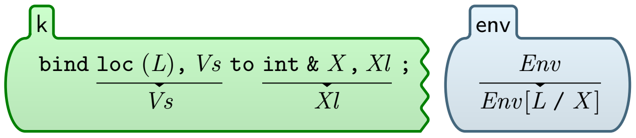
For call-by-value, the passed value V is stored into a new memory location which is bound to the formal parameter:

RULE BIND-CALL-BY-VALUE



For call-by-reference, the location pointed to by the l-value is directly bound to the formal parameter:

RULE BIND-CALL-BY-REFERENCE



Finally, once all parameters have been bound, the binding construct dissolves:

RULE

$$\frac{\texttt{bind}\bullet\texttt{to}\bullet\texttt{;}}{\bullet}$$

5 On the Semantics of a $\mathbb{K}$ Definition

This section briefly presents the transition semantics of a $\mathbb{K}$ definition. Understanding this semantics is essential for understanding the differences between $\mathbb{K}$ and rewriting logic and thus making correct use of the Maude tools to analyze the behavior of programs against the executable semantics of a language.

As pointed out in the previous section, a $\mathbb{K}$ definition consists of several components: a language syntax (which is a set of *KLabel* constants) possibly annotated with strictness and other attributes and possibly extended with additional syntactic constructs needed for semantic reasons, an initial configuration, and a set of rewrite rules. As seen, several of $\mathbb{K}$'s features are in fact just notations, allowing users to define more compact or more modular semantics. For example, the strictness annotations can be desugared in pairs of special heating/cooling rules, the configuration abstraction can complete the cell structure of rules to match that of the initial configurations, and so on. Then a natural question is what is $\mathbb{K}$, after all, from a theoretical, minimalistic perspective, and what is its semantics. In this section we address this question at an informal level, referring the interested reader to [2,23] for more technical details.

A $\mathbb{K}$ *definition* (or $\mathbb{K}$ *rewrite theory*, or $\mathbb{K}$ *rewrite system*, or even just a $\mathbb{K}$ *system*) is a triple (Σ, S, C), where Σ is an algebraic signature and where S and C are sets of $\mathbb{K}$ rewrite rules, the former called structural rules and the latter called computational rules. Σ includes operation symbols for all the desired language constructs, builtin data types and values, auxiliary operations needed for the semantics (e.g., `bind_to_`), operations corresponding to cells, operations corresponding to $\mathbb{K}$-tool-provided data-structures such as lists, sets, maps, etc.

The formal definition of a $\mathbb{K}$ *rule* is rather technical [2,23]. Intuitively, a $\mathbb{K}$ rule consists of a shared *pattern*, which is a multi-context with a distinguished hole for each underlined sub-term in the rule (i.e., sub-term that rewrites), together with two mappings of these special holes: one corresponding to the sub-terms above the line, and another to the sub-terms underneath the line. Any regular rewrite rule is a particular $\mathbb{K}$ rule with an empty pattern (i.e., just a hole). As shown in [2,23], a set R of $\mathbb{K}$ rules yields a *concurrent rewrite relation* $\Rrightarrow_R$ on Σ-terms. As expected, $\Rrightarrow_R$ can be serialized into sequences of ordinary rewrite steps obtained by turning each $\mathbb{K}$ rule into a regular rewrite rule by forgetting the shared pattern information, that is, by infusing the pattern into both the left-hand-side and the right-hand-side terms. Thus, $\Rrightarrow_R$ can and should be regarded as a more concurrent variant of rewriting, one which takes into account the specifics of the $\mathbb{K}$-rules, namely their capability to share resources.

The split of rules into structural S and computational C in a $\mathbb{K}$ definition (Σ, S, C) is purely methodological; there are no hard requirements on what should be structural and what should be computational. In general, we think of structural rules as rearranging the configuration before or after a computational rule applies. Besides heating/cooling rules and language-specific syntax desugarings, S typically also includes rules telling how the underlying mathematical domains or builtin libraries operate; since an equation can be regarded as two opposite rules, usual algebraic data types can also be captured by means of structural rules, and usual equational deduction can be mimicked with structural rewrites using S. As their name indicates, computational rules are the ones that count as computations. In terms of the generated transition system, the structural rules are not observable while the computational rules are observable.

Formally, given a $\mathbb{K}$ definition (Σ, S, C), we let $\Rrightarrow$ denote the relation $\Rrightarrow_S^* \circ \Rrightarrow_C \circ \Rrightarrow_S^*$. In other words, $\gamma \Rrightarrow \gamma'$ if and only if γ can be structurally rearranged into a term which is computationally transformed into a term which can be structurally rearranged into γ'. Or even simpler, γ rewrites to γ' using precisely one computational rule. The relation $\Rrightarrow$ associates a transition system to any term γ, which we can think of as the behavior of γ under the given $\mathbb{K}$ definition. Consider, for example, the initial configuration of CinK, say `cfg[`$\$PGM$`]` (assume $\$IN$ instantiated to some arbitrary input), and some CinK program P. Then *the* $\mathbb{K}$ *semantics of* P is the transition system associated to the configuration term `cfg[`P`]`, depicted in Figure 3: boxes enclose the structural rearrangements via $\Rrightarrow_S^*$, which appear as dashed arrows in the figure, and full arrows between boxes depict the relation $\Rrightarrow_C$. We can even let $[\![P]\!]$ denote this transition system.

Therefore, in addition to allowing rules that explicitly specify what can be concurrently shared with other rules, another major difference between $\mathbb{K}$ and rewriting logic is that $\mathbb{K}$ has no equations. Equations can be expressed in $\mathbb{K}$ as two opposite structural $\mathbb{K}$ rules with zero sharing. A question then is how to develop $\mathbb{K}$ tools that can effectively execute and formally analyze $\mathbb{K}$ definitions. Ideally, an implementation would statically analyze the rules in S and C, and make use of efficient decision procedures for common fragments of S (e.g., when it contains rules corresponding to equations such as associativity, commutativity, identity, etc.) and of specialized data-structures and even decision procedures for heating/cooling rules, and so on. Unfortunately, these seem hard. Our current

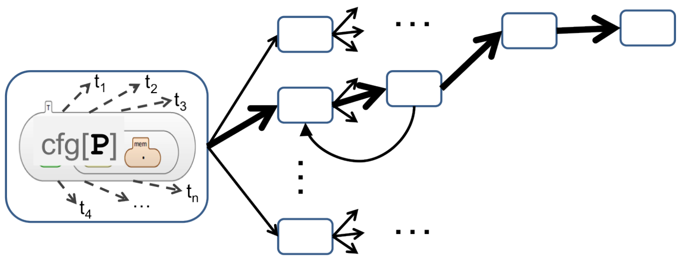


Fig. 3. The $\mathbb{K}$ transition system for the execution of a program P

approach in our $\mathbb{K}$ tool prototype is to (automatically) compile $\mathbb{K}$ definitions to Maude, allowing the user to intervene in the process. Specifically, the current version of the $\mathbb{K}$ tool compiles a $\mathbb{K}$ definition (Σ, S, C) into a Maude system module $(\Sigma^{\text{Maude}}, A, E, R)$, following the following rules:

- Each ground Σ-configuration is represented by a ground Σ^{Maude}-term;
- Each structural rule in S is compiled either into an axiom in A (e.g., associativity, commutativity, identity) or into a Maude equation in E;
- Each computational rule in C is compiled either into a Maude equation in E or into a Maude rewrite rule in R.

The $\mathbb{K}$ tool provides an annotation system by which the user can instruct the tool which computational rules are to be compiled into Maude equations and which into Maude rewrite rules (see Section 6 for examples).

An immediate advantage of compiling $\mathbb{K}$ definitions into Maude rewrite theories is that the $\mathbb{K}$ tool can be used as an interpreter, i.e., given a program P it can execute it according to the semantics of its language; the execution describes a path from the initial configuration to a normal form (irreducible configuration). Such an execution is intuitively represented by the thick arrows in Figure 3.

6 Executing and Analyzing $\mathbb{K}$ Definitions in Maude

In this section we describe how the $\mathbb{K}$ tool, taking advantage of the generic Maude tool suite, can be used to execute programs against the $\mathbb{K}$ definition of their language and to analyze their behavior.

The following command asks the $\mathbb{K}$ tool to compile the definition of CinK into a Maude module; we assume that the command is executed in a directory containing the definition of CinK in the `cink.k` file:

```
$ kompile cink
$ ls *.maude
cink-compiled.maude
```

6.1 Executing programs

Consider the following program:

```
int r;                          int main() {
int f(int x) {                    r = 5;
  return (r = x);                 return f(1) + f(2), r;
}                               }
```

Assuming this program is contained into a file `nondet.cink` in the `programs` directory, its execution can be obtained with the following command:

```
$ krun programs/nondet.cink
<T>
  <k> 1;  </k>
  ...
  <genv> ...   r |-> 0 </genv>
  <store> 0 |-> 1 ... </store>
</T>
```

The tool displays the final configuration reached on one of the execution paths, which contains the result of the computation. The information stored in cells is very useful when, e.g., the normal form is a dead-lock configuration. However, `krun` can be also used as an interpreter. For example, if we replace the last line from `main` with "`cout <<  f(1) + f(2) << r << "\n";`" and execute the program with the `no-config` option, we obtain:

```
$ krun --no-config programs/nondet.cink
3 1
```

The mechanism that connects a cell (here the `out` cell) to the standard input/output is described in [24].

6.2 Analyzing executions

The transition system associated to a $\mathbb{K}$ definition can be explored using two Maude tools: the search command and the LTL model checker.

Search. `krun` provides the `search` option to explore all execution paths starting from the initial configuration and display the final configurations obtained along these paths. The implementation of this options uses Maude's search engine. The command below displays all possible outcomes for the `nondet` program above:

```
$ krun nondet.cink --search
Search results:
Solution 1, state 0:
<T>
  <k> 1; </k>
  ...
  <genv>... r |-> 0 </genv>
  <store> 0 |-> 1 ...</store>
</T>
```

Although the behavior of this program is non-deterministic, only one solution is reported. This is a consequence of how the Maude module is generated from the 𝕂 definition. To obtain the full behavior of the above program, the tool must be instructed to compile the 𝕂 rules that are the source of non-determinism into Maude rewrite rules. Here the non-determinism is given by the evaluation order of the addition operator. We can use the annotation superheat for the addition operator to specify that its heating rules must generate Maude rewrite rules:

SYNTAX *Exp* ::= *Exp* + *Exp* [superheat strict]

Now the above command displays all executions paths:

```
$ krun nondet.cink --search
Search results:
Solution 1, state 1:
<T>
  <k> 1; </k>
  ...
  <genv>... r |-> 0 </genv>
  <store> 0 |-> 1 ...</store>
</T>
Solution 2, state 2:
<T>
  <k> 2; </k>
  ...
  <genv>... r |-> 0 </genv>
  <store> 0 |-> 2 ...</store>
</T>
```

This example shows that members of the same structural class can generate distinct, non-joinable transitions.

The cooling rules could have a similar effect. Using the 𝕂 tool to explore the behaviors of program

```
int print(int n) {                int main() {
  cout << n;                        (print(1) + print(2) + print(3));
  return n;                         return 0;
}                                 }
```

only displays four solutions. The cause is that, by default, cooling rules (e.g., those for the operands of +) are only applied when their arguments are reduced to values. However, if we annotate the rule for `return` on page 8 with the tag supercool, then the cooling rules will apply eagerly after the application of this rule, cooling the entire computation before attempting to heat it again, and thus allowing all 6 possible solutions to be observed.

The superheat and supercool tags described above are compiled to Maude so that they offer the 𝕂 tool user the following intuition: when a superheat operation is reached during the execution of the program, an "exhaustive non-determinism" mode is entered; when a supercool rule is applied, the next non-deterministic behavior is explored. This way, superheat/supercool act as user-defined begin/end brackets within the non-deterministic state-space of the program where exhaustive non-determinism is desired to be explored.

Another way to specify that certain rules should generate transitions into the transition system generated by Maude is to annotate them with the transition tag. If the definition of CinK is compiled without any transition tags, then the tool will explore only one execution for the following multi-threaded program:

```
int r;                          int main() {
int f(int x) {                    std::thread t1(f, 1);
  return (r = x);                 std::thread t1(f, 2);
}                                 return r;
                                }
```

However, if we annotate the rules for memory lookup and memory update with the transition tag, the tool will display all 8 possible execution outcomes.

At this stage, the reader may wonder why don't we automatically tag all the operations with superheat and all the rules, both structural and computational, with supercool, and all the computational rules also with transition. While this would indeed guarantee that no behaviors are lost in compilation, in our experience doing so typically yields impractical Maude definitions, whose state-space is too large to search. In general, most of the users of $\mathbb{K}$ are interested in fast execution first place, and only then, potentially, in searching. Thus, we decided that the default compilation of the $\mathbb{K}$ tool optimizes execution. Searching is considered expert use of the tool. Even experts typically start conservatively, by adding only one or two tags, and then increase their number depending on the complexity of the tested program, too, and only if performance is acceptable. It would be interesting to develop automatic criteria or techniques that provide guarantees of exhaustive behavior exploration with a limited number of tags, but this is beyond our scope here. The $\mathbb{K}$ tool currently provides no such criteria.

Model Checking. The $\mathbb{K}$ tool also includes a hook to Maude's LTL model-checker, where the latter's sorts and operations are renamed to avoid name clashes and to follow the $\mathbb{K}$ tool's convention for naming builtin items. For instance, the sort name for the transition system states is `#ModelCheckerState` and that for the atomic propositions is `#Prop`. For similar reasons, the operators for LTL formulas are prefixed with "LTL". As any builtin sort is subsorted to `K`, `#ModelCheckerState` is also a subsort of `K`.

The Maude module obtained by compiling a $\mathbb{K}$ definition is based on the abstract syntax tree (AST) representation of both the language constructs and the $\mathbb{K}$ constructs. Hence the direct use of Maude to define properties and to call the model-checker for a given initial configuration and a given LTL formula is not quite user-friendly. We created an interface to facilitate the use of the model-checker. We describe the use of this interface by means of a program describing the Dekker's algorithm (see Figure 4). Let us assume that we want to show that this program satisfies the LTL formula

$$\texttt{LTL[]}(\ \texttt{eqTo}(\texttt{critical1}, 1)\ \texttt{LTL->}\ \texttt{eqTo}(\texttt{critical2}, 0)),$$

where `LTL[]` denotes the always modal operator, `LTL->` the implication, and the atomic proposition `eqTo`(X, I) is satisfied by the current configuration if and only if the value of the variable X is equal to I. This property says that for each configuration reachable from the initial one the value of global variable `critical2` is equal to `0` whenever the value of `critical1` is equal to `1`, representing half of the mutual exclusion property (the other half is symmetrical).

```
int flag1 = 0,  flag2 = 0;
int critical1 = 0, critical2 = 0;
int turn = 1;
int dekker1() {
  while (true) {
    flag1 = 1; turn = 2;
    while((flag2 == 1) &&
             (turn == 2)) { }
    // Enter critical section
    critical1 = 1;
    // Critical stuff ...
    // Leave critical section
    critical1 = 0;
    flag1 = 0;
  }
}
```

```
int dekker2() {
  while (true) {
    flag2 = 1; turn = 1;
    while((flag1 == 1) &&
             (turn == 1)) { }
    // Enter critical section
    critical2 = 1;
    // Critical stuff ...
    // Leave critical section
    critical2 = 0;   flag2 = 0;
  }
}
int main() {
  std::thread t1(dekker1);
  std::thread t2(dekker2);
}
```

Fig. 4. Dekker's algorithm in CinK

We propose the following solution for model-checking this property. The LTL formulas are similar to programming languages: they have syntax and semantics. Therefore we define the syntax and the semantics in separate modules. For this example, the syntax module defines the atomic proposition `eqTo`:

MODULE CINK-PROP-SYNTAX

  IMPORTS CINK-SYNTAX

  IMPORTS LTL-HOOKS

  SYNTAX $\quad$ *#Prop* ::= `eqTo`(*Id*, *Val*)

END MODULE

Note the simplicity of this module. Generally, such a module should define the languages for properties to be checked for the defined language. The module LTL-HOOKS provides a $\mathbb{K}$ interface to the Maude module defining the syntax of LTL formulas.

The module for semantics has a simple structure, too:

MODULE CINK-PROP-SEMANTICS

  IMPORTS MODEL-CHECKER-HOOKS

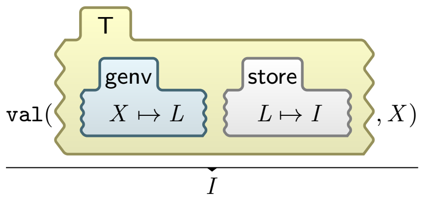
  IMPORTS CINK-PROP-SYNTAX

IMPORTS CINK-SEMANTICS

SYNTAX $\#ModelCheckerState$ ::= `KItem`(Bag)

SYNTAX Int ::= `val`(Bag, Id)

RULE



RULE

$\underbrace{\texttt{KItem}(B)\ \texttt{LTL|=}\ \texttt{\#eqTo}(X, I)}_{\texttt{true}}$

when `val`(B, X) $==_K$ I

END MODULE

The module MODEL-CHECKER-HOOKS provides a $\mathbb{K}$ interface to the Maude module implementing the model-checker algorithm. The sort for configurations is `Bag` and therefore it is injected as a subsort of `#ModelCheckerState`. The auxiliary function `val` (C, X) returns the value of the variable X in the configuration C. Note that its definition is given by just one rule. The last rule in the module gives the LTL semantics to the atomic proposition `eqTo`.

The definition of CinK together with these new modules is compiled, and then the `krun` command with the `--check` option is executed:

```
$ krun dekker.cink --check LTL[] (eqTo(critical1, 1) LTL->
                                   eqTo(critical2, 0))
```

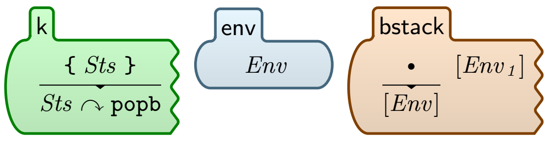
Remark 1. The full implementation of this command is in progress. For instance, when the formula is false the counter-example is huge. Currently, the output obtained from the corresponding Maude model checking command is displayed unformatted—we are working on finding a nicer way to represent it.

The above command works fine if the following two conditions are fulfilled:

1. the definition includes enough annotations to generate a transition system representing a faithful abstraction of the intended $\mathbb{K}$ semantics;
2. the set of configurations reachable from the initial configuration is finite.

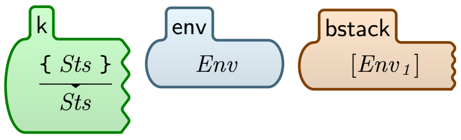
Unfortunately, the second condition is not satisfied by the initial configuration of the program describing the Dekker's algorithm. Since in CinK we may have variable declarations inside of blocks, each time the execution enters a block, the environment is saved in the cell `bstack`. Hence the two infinite `while` loops will infinitely increase the size of this cell during the exploration process. A small change of this rule re-establishes the needed property. The `while` loop does not include declarations of variables, so saving the environment is useless. We modify the semantics of the block statement such that the environment is saved only if it differs from the one stored in the top of the cell:

RULE BLOCK



when Env =/=$_{Map}$ Env_1

RULE BLOCK



when Env ==$_{Map}$ Env_1

Remark 2. The search command and the model checker must be carefully used since, as we already mentioned, the Maude modules produced by the 𝕂 tool are not always faithfully representing the intended 𝕂 transition system. The main motivation for this choice is given by efficiency. The user can use the annotations (tags) to guide the compilation process into obtaining good abstraction of the 𝕂 transition system. However, even if the Maude transition system is a good abstraction of the 𝕂 one, it often is a (strict) subsystem of that giving the transition semantics to the original 𝕂 definition.

7 Conclusions

This paper gave a high-level overview of the 𝕂 framework: its motivation and objective, what it is and how it works, and its relationship to rewriting logic and Maude. The 𝕂 framework and the 𝕂 tool have by now reached maturity, and are currently being actively used for defining real programming languages and experimenting with various language features. Besides didactic and prototypical languages (such as lambda calculus, System F, and Agents), the 𝕂 tool was used to formalize C [25] (and to analyze C programs [26]) and Scheme [27]; additionally, definitions of Haskell, Javascript, X10, a framework for domain specific languages [28,29] or P-Systems [30], a RISC assembly language [31], and LLVM are underway. With respect to analysis tools, the 𝕂 tool was used for tools like type checkers and type inferencers [32], and in the development of a new deductive program verification tool using program assertions based on matching logic [33,34], model checking tools [35,36], symbolic execution [37,38], computing worst case execution times [39], or researching runtime verification techniques [40,23]. All these definitions and analysis tools can be found on the 𝕂 framework website at `http://k-framework.org`.

References

1. Roșu, G.: CS322, Fall 2003 - Programming Language Design: Lecture Notes. Technical Report UIUCDCS-R-2003-2897, University of Illinos at Urbana Champaign (December 2003) Lecture notes of a course taught at UIUC.
2. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6) (2010) 397–434
3. Serbanuta, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Rosu, G.: The K primer (version 2.5). In Hills, M., ed.: Proceedings of the Second International K Workshop, K'11. Volume to appear of Electronic Notes in Theoretical Computer Science. (2012)
4. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude, A High-Performance Logical Framework. Volume 4350 of Lecture Notes in Computer Science. Springer (2007)
5. Meseguer, J., Roșu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In Basin, D.A., Rusinowitch, M., eds.: Automated Reasoning - Second International Joint Conference, IJCAR 2004, Proceedings. Volume 3097 of Lecture Notes in Computer Science., Springer (2004) 1–44
6. Meseguer, J., Roșu, G.: The rewriting logic semantics project. Theoretical Computer Science **373**(3) (2007) 213–237
7. Meseguer, J., Rosu, G.: The rewriting logic semantics project: A progress report. In Owe, O., Steffen, M., Telle, J.A., eds.: FCT. Volume 6914 of Lecture Notes in Computer Science., Springer (2011) 1–37
8. Lucanu, D., Șerbănuță, T.F.: Cink - an exercise of thinking in K. Technical Report TR12-03, Department of Computer Science, Alexandru Ioan Cuza University of Iaşi (2012) `http://thor.info.uaic.ro/~tr/tr.pl.cgi`.
9. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1) (1992) 73–155
10. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science **403**(2-3) (2008) 239–264
11. Farzan, A., Chen, F., Meseguer, J., Roșu, G.: Formal analysis of Java programs in JavaFAN. In Alur, R., Peled, D., eds.: Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. Volume 3114 of Lecture Notes in Computer Science., Springer (2004) 501–505
12. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. STTT **2**(4) (2000) 366–381
13. Meseguer, J.: Rewriting logic as a semantic framework for concurrency: a progress report. In Montanari, U., Sassone, V., eds.: CONCUR'96. Volume 1119 of Lecture Notes in Computer Science., Springer (1996) 331–372
14. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude 2. In Gadducci, F., Montanari, U., eds.: Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002). Volume 71 of Electronic Notes in Theoretical Computer Science., Elsevier (2002)
15. Șerbănuță, T.F., Roșu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Information and Computation **207** (2009) 305–340
16. Kahn, G.: Natural semantics. In Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M., eds.: STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings. Volume 247 of Lecture Notes in Computer Science., Springer (1987) 22–39

17. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61** (2004) 17–139 Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.
18. Mosses, P.D.: Modular structural operational semantics. Journal of Logic and Algebraic Programming **60-61** (2004) 195–228
19. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94
20. Friedman, D.P., Wand, M., Haynes, C.T.: Essentials of Programming Languages. 2nd edn. MIT Press, Cambridge, MA (2001)
21. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science **96**(1) (1992) 217–248
22. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark (November 2004) This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.
23. Şerbănuţă, T.F.: A Rewriting Approach to Concurrent Programming Language Design and Semantics. PhD thesis, University of Illinois at Urbana-Champaign (December 2010) `https://www.ideals.illinois.edu/handle/2142/18252`.
24. Arusoaie, A., Şerbănuţă, T.F., Ellison, C., Roşu, G.: Making maude definitions more interactive. In: Rewriting Logic and Its Applications, WRLA 2012, Springer (2012) this volume
25. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12), ACM (2012) 533–544
26. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: 33nd Conference on Programming Language Design and Implementation (PLDI'12), ACM (2012) to appear.
27. Meredith, P., Hills, M., Roșu, G.: An executable rewriting logic semantics of k-scheme. In Dube, D., ed.: Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701, Laval University (2007) 91–103
28. Rusu, V., Lucanu, D.: A K-based formal framework for domain-specific modelling languages. In: 2nd International Conference on Formal Verification of Object-Oriented Software. (2011) 306–323
29. Rusu, V., Lucanu, D.: K semantics for OCL—a proposal for a formal definition for OCL. In Hills, M., ed.: Proceedings of the Second International K Workshop, K'11. Volume to appear of Electronic Notes in Theoretical Computer Science. (2012)
30. Șerbănuță, T.F., Stefanescu, G., Roșu, G.: Defining and executing P systems with structured data in K. In Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A., eds.: Workshop on Membrane Computing (WMC'08). Volume 5391 of Lecture Notes in Computer Science., Springer (2009) 374–393
31. Asavoae, M.: K semantics for assembly languages: A case study. In Hills, M., ed.: Proceedings of the Second International K Workshop, K'11. Volume to appear of Electronic Notes in Theoretical Computer Science. (2012)
32. Ellison, C., Șerbănuță, T.F., Roșu, G.: A rewriting logic approach to type inference. In: Recent Trends in Algebraic Development Techniques — 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers. Volume 5486 of Lecture Notes in Computer Science., Springer (2009) 135–151

33. Roşu, G., Ellison, C., Schulte, W.: Matching logic: An alternative to Hoare/Floyd logic. In Johnson, M., Pavlovic, D., eds.: Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10). Volume 6486 of Lecture Notes in Computer Science. (2010) 142–162
34. Roşu, G., Ştefănescu, A.: Matching logic: A new program verification approach (NIER track). In: 30th International Conference on Software Engineering (ICSE'11). (2011) 868–871
35. Asavoae, I.M., Asavoae, M.: Collecting semantics under predicate abstraction in the k framework. In Ölveczky, P.C., ed.: WRLA. Volume 6381 of Lecture Notes in Computer Science., Springer (2010) 123–139
36. Asavoae, I.M., de Boer, F., Bonsangue, M.M., Lucanu, D., Rot, J.: Bounded model checking of recursive programs with pointers in k. In: WRLA. (2012) as extended abstract in the ETAPS preproceedings.
37. Asavoae, I.M., Asavoae, M., Lucanu, D.: Path directed symbolic execution in the k framework. In Ida, T., Negru, V., Jebelean, T., Petcu, D., Watt, S.M., Zaharie, D., eds.: SYNASC, IEEE Computer Society (2010) 133–141
38. Asavoae, I.M.: Abstract semantics for alias analysis in K. In Hills, M., ed.: K'11. Electronic Notes in Theoretical Computer Science (2012) to appear.
39. Asăvoae, M., Lucanu, D., Roşu, G.: Towards semantics-based WCET analysis. In: WCET. (2011)
40. Roșu, G., Schulte, W., Șerbănuță, T.F.: Runtime verification of C memory safety. In: Runtime Verification (RV'09). Volume 5779 of Lecture Notes in Computer Science. (2009) 132–152