

Mapping Modular SOS to Rewriting Logic

Christiano de O. Braga¹, E. Hermann Hæusler²,
José Meseguer³, and Peter D. Mosses⁴

¹ Departamento de Ciência da Computação, Universidade Federal Fluminense

² Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro

³ Computer Science Department, University of Illinois at Urbana-Champaign

⁴ BRICS & Department of Computer Science, University of Aarhus

Abstract. Modular SOS (*MSOS*) is a framework created to improve the modularity of structural operational semantics specifications, a formalism frequently used in the fields of programming languages semantics and process algebras. With the objective of defining formal tools to support the execution and verification of *MSOS* specifications, we have defined a mapping, named *MtoR*, from *MSOS* to rewriting logic (*RWL*), a logic which has been proposed as a logical and semantic framework. We have proven the correctness of *MtoR* and implemented it as a prototype, the MSOS-SL Interpreter, in the Maude system, a high-performance implementation of *RWL*. In this paper we characterize the *MtoR* mapping and the MSOS-SL Interpreter.

The reader is assumed to have some basic knowledge of structural operational semantics and object-oriented concepts.

1 Introduction

In this paper we present a formal mapping from modular structural operational semantics (*MSOS*) to rewriting logic (*RWL*), named *MtoR*, its correctness proof, and its prototype implementation, the MSOS-SL Interpreter.

The intuitive idea underlying the *MtoR* mapping is that there is a very close connection between *SOS* transition rules and rewrite rules in rewriting logic: each *SOS* transition rule becomes a *conditional rewrite rule*, with the premises of the *SOS* transition rule translated into *conditions* of the rewrite rule, and the conclusion of the *SOS* transition rule translated into the rewrite rule itself. Furthermore, modularity in *MSOS* transition rules is preserved by translating it as *inheritance* in object-oriented rewrite theories [8].

This work makes these intuitions precise by defining formally the *MtoR* map and proving its correctness. Using reflection the *MtoR* map can then be defined in an executable way in the Maude *RWL* interpreter, thus giving rise to the MSOS-SL Interpreter.

This paper is organized as follows. Section 2 briefly presents the *MSOS* and *RWL* frameworks. Section 3 formally describes the *MtoR* mapping, showing the syntactical transformation from *MSOS* specifications to *RWL* rewrite theories and the semantic mapping from arrow-labeled transition systems (*ALTS*), the models of *MSOS* specifications, to *R*-systems, the models of rewrite theories. Section 4 presents the proof

of correctness of $\mathcal{M}to\mathcal{R}$, showing that computations are preserved in the model of the generated rewrite theory. Section 5 briefly describes the MSOS-SL Interpreter, our prototype implementation of $\mathcal{M}to\mathcal{R}$. We conclude this paper in Section 6 with some final remarks and future work.

2 Modular SOS and Rewriting Logic

This section formally presents the *MSOS* and *RWL* frameworks. Section 2.1 begins with a motivation for *MSOS* giving a small example for the modularity problem in *SOS* and then formalizes the concepts of a *MSOS* specification and an arrow-labeled transition system, the model for a *MSOS* specification. Section 2.2 gives very concise definitions for rewrite theories in rewriting logic and their models, viewed as transition systems. These two definitions are used in the formal characterization of the $\mathcal{M}to\mathcal{R}$ mapping in Section 3 and in the correctness proof of $\mathcal{M}to\mathcal{R}$ in Section 4, respectively.

2.1 Modular Structural Operational Semantics

Modularity in *SOS* specifications was left as an open problem by Plotkin in [14]. To illustrate the *SOS* modularity problem, let us consider the following *SOS* rules for the gradual evaluation of expressions e to their values v , where an expression is an addition of expressions or a value, and values are natural numbers.

Example 1 (Addition of natural numbers in SOS).

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \quad \frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2} \quad \frac{e_2 \rightarrow e'_2}{v_1 + e_2 \rightarrow v_1 + e'_2} \quad (1)$$

If one now considers a functional extension, the existing rules would have to be modified in order to support the notion of environments, besides the addition of the rule for function application. In the original formulation of the rules, the transition relation only involved expressions as configurations. Now it involves both expressions and an environment, so all the rules have to be reformulated.

A further imperative extension would cause more modifications, adding the notion of stores. The transition relation now involves the syntax, the environment, and the store.

Extensions with (interactive) input-output would require the use of labeled transitions, and entail yet another reformulation of the above rules. We refer to [5, 12] for the complete presentation of Example 1 and a second example describing an extension to the ML language with concurrency primitives, respectively.

To improve the modularity of *SOS* specifications, and in particular to avoid the need for reformulations, *MSOS* restricts configurations to (abstract) syntax, and requires semantic structures (such as environments and stores) to be incorporated in labels on transitions.

In categorical terms, the labels can be seen as the arrows of a category where the semantic states are the objects. For this reason such a transition system is called an arrow-labeled transition system (*ALTS*).

Definition 1 (Arrow-labeled transition systems). *An arrow-labeled transition system or ALTS is a labeled transition system where its set of labels is the set of the arrows of a category \mathbb{A} , represented as $\text{Morph}(\mathbb{A})$. The objects of \mathbb{A} , written $\text{Obj}(\mathbb{A})$, are the states of the ALTS. The source $\text{pre}(\alpha)$ of a label α on a transition, together with the configuration “before” the transition, represents the state of the computation before the transition, and its target $\text{post}(\alpha)$, together with the configuration “after” the transition, represents the state afterward. The composition of arrows α_1 and α_2 is defined iff $\text{post}(\alpha_1) = \text{pre}(\alpha_2)$. The identity arrows of \mathbb{A} are written $\mathbb{I}^{\mathbb{A}}$, or simply \mathbb{I} when \mathbb{A} is evident; they are called silent or unobservable arrows and are used to label transitions that merely reduce configurations. For two transitions $\gamma \xrightarrow{\alpha_1} \gamma', \gamma'' \xrightarrow{\alpha_2} \gamma'''$ to be adjacent, two conditions must be met, namely $\gamma' = \gamma''$, and $\text{post}(\alpha_1) = \text{pre}(\alpha_2)$. Terminating or infinite sequences of adjacent transitions then define the computations of the ALTS.*

In the context of the *MSOS* specification of a programming language semantics, a label represents the “semantic” state of a program, that is, the information associated, for example, with the store or with the environment. A label captures the semantic state both before and after a transition, possibly with some more information associated with the transition itself (such as the usual synchronization signals of CCS [10]).

Label categories are built using the so called *label transformers*. Label transformers assume that the label category \mathbb{A} comes equipped with additional structure, such that the arrows in \mathbb{A} have components. Each component is accessed by using a particular index in a set *Index* of indices, which is mapped to a value in the set *Univ* of information structures. That is, it is assumed that there are sets *Index*, of indices, and *Univ*, of information structures, and functions $\text{set} : \text{Morph}(\mathbb{A}) \times \text{Index} \times \text{Univ} \rightarrow \text{Morph}(\mathbb{A})$ and $\text{get} : \text{Morph}(\mathbb{A}) \times \text{Index} \rightarrow \text{Univ}$, where the intuitive meaning is that $\text{set}(\alpha, i, u)$ changes the component of $\alpha \in \text{Morph}(\mathbb{A})$ indexed by i to u , leaving the rest unchanged; and $\text{get}(\alpha, i)$ yields the component of α indexed by i . Thus, labels might be seen as *mappings* from indices to *Univ*.

A label transformer T involves adding a fresh new index, say i , to the original set *Index*, and transforms the original category of labels \mathbb{A} to the product category $T(\mathbb{A}) = \mathbb{A} \times \mathbb{C}_T$. The original set and get functions are extended in the obvious way, that is, by treating the arrows $\beta \in \mathbb{C}_T$ as the information stored in the new index i .

There are three such label transformers that can be used as basic building blocks to extend the label categories of *MSOS* specifications. They correspond to the following three basic choices of \mathbb{C}_T :

- A discrete category E , typically understood as a set of environments, giving rise to the label transformer **ContextInfo**(i, E).
- A cartesian product S^2 , understood as a binary relation, and viewed as a category with set of objects S and arrow composition given by relational composition. S is typically understood as a set of stores. This gives rise to the label transformer **MutableInfo**(i, S). The functions get_pre and set_post are defined to access the first, respectively change the second, component of the pair.
- A monoid (A, f, τ) , viewed as a one-object category, and is typically understood as a set of actions. The **EmittedInfo**(i, A, f, τ) label transformer arises with this choice of \mathbb{C}_T .

Definition 2 (MSOS specifications). A specification in modular structural operational semantics (MSOS) is a structure of the form $\mathcal{M} = \langle \Sigma, Lc, Tr \rangle$. The Σ component is the signature of \mathcal{M} , defined as a set of operation signatures characterizing the language defined by \mathcal{M} . The Lc component is a label category declaration, named Lc , and is defined as an application of a composition of any combination of the three label transformers to a label category. The component Tr is the set of transition rules of \mathcal{M} , represented as inference rules¹.

Example 2 (MSOS for the language in Example 1). The labeled transition rules of the MSOS specification for the language in Example 1 considering the functional and imperative extensions that we have described are as follows:

$$\frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\iota} n} \quad \frac{e_1 \xrightarrow{\alpha} e'_1}{e_1 + e_2 \xrightarrow{\alpha} e'_1 + e_2} \quad \frac{e_2 \xrightarrow{\alpha} e'_2}{v_1 + e_2 \xrightarrow{\alpha} v_1 + e'_2} \quad (2)$$

$$\frac{e_1 \xrightarrow{\alpha} e'_1}{e_1 e_2 \xrightarrow{\alpha} e'_1 e_2} \quad \frac{e_2 \xrightarrow{\alpha} e'_2}{v_1 e_2 \xrightarrow{\alpha} v_1 e'_2} \quad (3)$$

$$\frac{\alpha' = \mathbf{set}(\alpha, env, \mathbf{get}(\alpha, env)[x \mapsto v]) \quad e \xrightarrow{\alpha'} e'}{\lambda x(e) v \xrightarrow{\alpha} e'} \quad (4)$$

$$\frac{\mathbf{get}(\iota, env)(x) = v}{x \xrightarrow{\iota} v} \quad (5)$$

$$\frac{\alpha = \mathbf{set_post}(\iota, sto, \mathbf{get_pre}(\iota, sto)[l \mapsto v])}{l := v \xrightarrow{\alpha} ()} \quad (6)$$

$$\frac{\mathbf{get_pre}(\iota, sto)(l) = v}{l \xrightarrow{\iota} v} \quad (7)$$

The label category may be defined by the following nested application of label transformers:

$$\mathbf{MutableInfo}(sto, Store)(\mathbf{ContextInfo}(env, Env)(\mathbb{1})). \quad (8)$$

That is, $Index = \{env, sto\}$ corresponding to adding two new components to the trivial category $\mathbb{1}$ (consisting of just one object and one arrow), namely the arrows and the objects of the categories formed from the sets Env and $Store$ by the label applied transformers.

Note that the above rules did not require any change to accommodate the functional and imperative extensions, and that they would not have to be changed at all in a further extension. To allow concurrency and message-passing, for instance, as in primitives [12], one would add a new semantic structure to the labels. The new rules would set and get data to and from the new index involved in the extension, but the previous rules wouldn't have to be changed.

¹ We refer to [4, Definition 4, page 6] for the grammar definition of the transition rules in an MSOS specification.

2.2 Rewriting Logic

Let us now present the rewriting logic framework by defining the concepts of rewrite theories (specifications in *RWL*) and \mathcal{R} -systems, the models for rewrite theories. We refer to [9] for a complete presentation of the rewriting logic framework.

Definition 3 (Rewrite theory). A rewrite theory \mathcal{R} is a 4-tuple $\mathcal{R} = \langle \Sigma, E, L, R \rangle$ where Σ is a ranked alphabet of function symbols, E is a set of Σ -equations, L is a set of labels, and R is a set of pairs $R \subseteq L \times (T_{\Sigma, E}(X)^2)^+$ whose first component is a label and whose second component is a nonempty sequence of pairs of the E -equivalence class of terms with $X = \{x_1, \dots, x_n, \dots\}$ a countably infinite set of variables. Elements of R are called rewrite rules.

Due to space limitations, let us briefly explain the syntax for object-oriented rewrite theories in rewriting logic necessary to follow the examples. A *class* is declared in a rewrite theory using the syntax *class* $\langle \text{class-name} \rangle \mid \langle \text{attribute-name} \rangle : \langle \text{attribute-type} \rangle, \dots \langle \text{attribute-name} \rangle : \langle \text{attribute-type} \rangle$. where the non-terminals *class-name*, *attribute-name*, and *attribute-type* are strings representing what their names imply. In an object-oriented rewrite theory an *object* is a term of sort *Object* structured according to the class declaration of the object's *class-name*. Object-oriented theories are actually “syntactic-sugar” for rewrite theories. Roughly speaking class declarations are mapped to sort declarations and inheritance is represented via the subsorting relation. We refer to [6] for a detailed presentation of the translation from object-oriented theories to rewrite theories.

A (conditional) rewrite rule is declared with the concrete syntax *cr1* : $\langle \text{left-hand-side-pattern} \rangle \Rightarrow \langle \text{right-hand-side-pattern} \rangle$ *if* $\langle \text{condition-set} \rangle$. The non-terminals *left-hand-side-pattern* and *right-hand-side-pattern* are nonempty sequences of the E -equivalence class of terms with variables. Roughly speaking, the condition set is a conjunction of predicates or rewrites.

Operations are defined using the keyword *op* and equations with the *eq* operator. Variables are declared using the keywords *var* or *vars*.

The paper [9] gives a precise definition for \mathcal{R} -systems in categorical terms. Instead, in Section 4 we use the computational view of \mathcal{R} -systems as transition systems to prove the correctness of $\mathcal{M}to\mathcal{R}$, the mapping from *MSOS* to *RWL*. Under the computational view the objects of a \mathcal{R} -systems \mathcal{S} are the states of the transition system \mathcal{TS} that models a rewrite theory \mathcal{R} and the arrows of \mathcal{S} are the transitions of \mathcal{TS} .

A very important characteristic its logic is *reflective* [1,3], in a precise mathematical way. There is a finitely presented rewrite theory \mathcal{U} that is *universal* in the sense that it can be represented in \mathcal{U} any finitely presented rewrite theory \mathcal{R} (including \mathcal{U} itself) as a term $\overline{\mathcal{R}}$, which is a *meta* representation of the rewrite theory \mathcal{R} ; any terms t and t' in \mathcal{R} as terms \overline{t} , $\overline{t'}$, which are the meta representations of terms t and t' respectively; and any pair (\mathcal{R}, t) as a term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, in such a way that the following equivalence holds

$$\mathcal{R} \vdash t \longrightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle. \quad (9)$$

Since \mathcal{U} is representable in itself, a “reflective tower” can be achieved with an arbitrary number of levels of reflection, since the following equivalences hold.

$$\mathcal{R} \vdash t \longrightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \dots$$

Reflection in rewriting logic gives the formal basis for the transformation process in $\mathcal{M}to\mathcal{R}$, since the transformation process is defined as a meta-function in rewriting logic, as explained in Section 3. This then allows a very direct reflective implementation of the MSOS-SL Interpreter in Maude.

3 $\mathcal{M}to\mathcal{R}$: Mapping MSOS to RWL

This section formally presents the mapping from modular structural operational semantics (*MSOS*) to rewriting logic (*RWL*).

Section 3.1 gives a formal definition for the *syntactical* transformation from *MSOS* specifications to *RWL* rewrite theories, represented by the function $\mathcal{M}to\mathcal{R} : \mathcal{M} \rightarrow \mathcal{R}$, and some auxiliary definitions. Section 3.2 formalizes the *semantic* mapping from arrow-labeled transition systems to \mathcal{R} -systems via the mapping from *ALTS* to transition systems (represented by the mapping $\mathcal{A}to\mathcal{T}\mathcal{S} : \mathcal{A} \rightarrow \mathcal{T}\mathcal{S}$), and the correspondence between transition systems and \mathcal{R} -systems in [9].

3.1 From MSOS Specifications to Rewrite Theories

In order to define a mapping from one language to another, one has to relate their syntaxes, and semantic models. In this section we define how *MSOS* specifications are transformed into rewrite theories. Actually we use *object-oriented* rewrite theories [6, 8], which provide some auxiliary syntactic sugar to specify classes and objects.

The main idea of the mapping, which is reflected both at syntactic and semantic levels, is to expose the information encapsulated in the labels of the label category into the configurations of rules, at the syntactic level, and to the computations states, at the semantic level.

At the syntactic level, the transformation of an *MSOS* specification \mathcal{M} maps the label category declaration of \mathcal{M} into a class declaration in a rewrite theory \mathcal{R} , and the transition rules of \mathcal{M} into rewrite rules in \mathcal{R} , such that the label information in *MSOS* transition rules is moved to the configurations of the *RWL* rewrite rules. The label variables and operations on labels on the premises of the transition rules of \mathcal{M} , are transformed into variables of sort *Object* and operations on objects, respectively, in \mathcal{R} .

The transformation for *MSOS* specifications is given in Definition 4. Note that the function $\mathcal{M}to\mathcal{R}$ is overloaded for each component in an *MSOS* specification, returning its counterpart in a rewrite theory.

Definition 4 (Transformation for MSOS specifications). *The transformation function $\mathcal{M}to\mathcal{R} : \mathcal{M} \rightarrow \mathcal{R}$, transforms an MSOS specification $\mathcal{M} = \langle \Sigma, Lc, Tr \rangle$ into a rewrite theory $\mathcal{R} = \langle \Sigma, R, E \rangle^2$ in the following way:*

- *The label category declaration Lc in \mathcal{M} is transformed into E , the equational part of the rewrite theory \mathcal{R} in the form of a class declaration named Lc , together with **get- i** and **set- i** operations on objects for each index i in Lc , with the expected*

² Labels in \mathcal{R} are abstracted to simplify the presentation.

behavior. (The operation **set-*i*** updates the value mapped to *i* with a given value, keeping the remaining attributes unchanged, and the operation **get-*i*** yields the value mapped to the attribute *i*.) Each index *i* in *Lc* become an attribute in *Lc*, typed according to the label transformer declaration that declared *i* in *Lc*, in the following way:

- For an index *i* declared in *Lc* via a **ContextInfo**(*i*, *T*) declaration, the type of the values mapped to *i* in *Lc* is *T*.
 - For an index *i* declared in *Lc* via a **MutableInfo**(*i*, *T*) declaration, the type of the values mapped to *i* in *Lc* is *T*.
 - For an index *i* declared in *M* via an **EmittedInfo**(*i*, *T*^{*}, ·, *e*), the type of the values mapped to *i* in *Lc* is *List*[*T*^{*}].³
- Each transition rule *tr* in *M*, is transformed into a rewrite rule *r* in *R* with left-hand side formed by a pair of the respective *tr* configuration and the pre label component. The right-hand side pattern of *r* is built analogously. We refer to [4, Sections 3.1 and 6.3.3] for a complete description of the transformation of the label operations. Allow α to be the label of the transition being specified, and β_1 to β_n label variables; $\sigma, \sigma', \gamma_1$ to γ_n, γ'_1 to γ'_n to be non-empty sequences of Σ -terms with variables; $\alpha_{\text{pre}}, \alpha_{\text{post}}, \beta_{1\text{pre}}$ to $\beta_{n\text{pre}}$ and $\beta_{1\text{post}}$ to $\beta_{n\text{post}}$ to be variables of type Object.

$$\mathcal{M}to\mathcal{R} \left(\frac{\gamma_1 \xrightarrow{\beta_1} \gamma'_1 \wedge \gamma_2 \xrightarrow{\beta_2} \gamma'_2 \dots \wedge \gamma_n \xrightarrow{\beta_n} \gamma'_n}{\sigma \xrightarrow{\alpha} \sigma'} \right) = \quad (10)$$

$$crl : (\sigma, \alpha_{\text{pre}}) \Rightarrow (\sigma', \alpha_{\text{post}})$$

$$if (\gamma_1, \beta_{1\text{pre}}) \Rightarrow (\gamma'_1, \beta_{1\text{post}}) \wedge \dots \wedge (\gamma_n, \beta_{n\text{pre}}) \Rightarrow (\gamma'_n, \beta_{n\text{post}}).$$

Premises that do not care about labels remain untouched. Identity-labeled transition rules are transformed into rewrite rules as described above together with the assertion that the associated **pre** and **post** variables of sort Object have to be equal.

Example 3 (The rewrite theory for Example 2).

The transformation of the label category yields a class declaration with two attributes, namely *sto* of type *Store* and *env* of type *Env*. In this example we will name the class *Example 2*, but in MSOS-SL (Section 5) a label declaration has an associated name which becomes the class name in the resulting rewrite theory.

$$\mathcal{M}to\mathcal{R}(\mathbf{MutableInfo}(sto, Store)(\mathbf{ContextInfo}(env, Env)(\mathbb{1}))) = \quad (11)$$

$$class \text{Example2} \mid sto : Store, env : Env .$$

The transformation of the label category declaration also generates set and get operation declarations and equations to *env* and *sto* object attributes, instances of class *Example2* and the appropriate variable declarations.

³ The reason why the type of the values mapped to **EmittedInfo**-declared indices are mapped to lists in the rewrite theory is because it is necessary to keep the “history” of the emitted information in a computation, due to the mapping of the label category to a *preorder*. This is a semantic issue which will become clearer in Definition 5.

The declaration for the label variable α gives rise to the declarations of the (mathematical) variables of sort *Object* α_{pre} and α_{post} .

The first rule in (2), which specifies a transition with an identity label, is transformed into a conditional rewrite rule as follows.

$$\begin{aligned} \mathcal{MtoR} \left(\frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\iota} n} \right) = & \quad (12) \\ \text{crl} : (n_1 + n_2, \iota_{\text{pre}}) \Rightarrow (n, \iota_{\text{post}}) \text{ if } n = n_1 + n_2 \wedge \iota_{\text{pre}} = \iota_{\text{post}} . \end{aligned}$$

The second rule in (2) is transformed into a conditional rewrite rule as follows.

$$\begin{aligned} \mathcal{MtoR} \left(\frac{e_1 \xrightarrow{\alpha} e'_1}{e_1 + e_2 \xrightarrow{\alpha} e'_1 + e_2} \right) = & \quad (13) \\ \text{crl} : (e_1 + e_2, \alpha_{\text{pre}}) \Rightarrow (e'_1 + e_2, \alpha_{\text{post}}) \text{ if } (e_1, \alpha_{\text{pre}}) \Rightarrow (e'_1, \alpha_{\text{post}}) . \end{aligned}$$

The third rule in (2) together with the rules in (3) have a similar transformation.

The transformation for the rule in (4), which specifies the application of a lambda abstraction, generates the following conditional rewrite rule.

$$\begin{aligned} \mathcal{MtoR} \left(\frac{\alpha' = \text{set}(\alpha, \text{env}, \text{get}(\alpha, \text{env})[x \mapsto v]) \quad e \xrightarrow{\alpha'} e'}{\lambda x (e) v \xrightarrow{\alpha} e'} \right) = & \quad (14) \\ \text{crl} : (\lambda x (e) v, \alpha_{\text{pre}}) \Rightarrow (e', \alpha_{\text{post}}) \text{ if} \\ \alpha'_{\text{pre}} = \text{set-env}(\alpha_{\text{pre}}, \text{get-env}(\alpha_{\text{pre}})[x \mapsto v]) \wedge \\ (e, \alpha'_{\text{pre}}) \Rightarrow (e', \alpha'_{\text{post}}) \wedge \\ \alpha_{\text{post}} = \text{set-env}(\alpha'_{\text{post}}, \text{get-env}(\alpha_{\text{pre}})[x \mapsto v]) . \end{aligned}$$

The rule in (5) specifies access to a value bound to a variable in the environment. It is transformed into the following conditional rewrite rule.

$$\begin{aligned} \mathcal{MtoR} \left(\frac{\text{get}(\iota, \text{env})(x) = v}{x \xrightarrow{\iota} v} \right) = & \quad (15) \\ \text{crl} : (x, \iota_{\text{pre}}) \Rightarrow (v, \iota_{\text{post}}) \text{ if } \iota_{\text{pre}} = \iota_{\text{post}} \wedge v = \text{get-env}(\iota_{\text{pre}})(x) . \end{aligned}$$

The rule in (6) specifies assignment of a value to a cell mapped to a variable in the storage. It is transformed into the following rewrite rule.

$$\begin{aligned} \mathcal{MtoR} \left(\frac{\alpha = \text{set_post}(\iota, \text{sto}, \text{get_pre}(\iota, \text{sto})[l \mapsto v])}{l := v \xrightarrow{\alpha} ()} \right) = & \quad (16) \\ \text{crl} : (l := v, \alpha_{\text{pre}}) \Rightarrow ((), \alpha_{\text{post}}) \text{ if} \\ \iota_{\text{pre}} = \alpha_{\text{pre}} \wedge \iota_{\text{pre}} = \iota_{\text{post}} \wedge \\ \alpha_{\text{post}} = \text{set_sto}(\iota_{\text{pre}}, \text{get_sto}(\iota_{\text{pre}})[l \mapsto v]) . \end{aligned}$$

Finally, the rule in (7) specifies how to access a value stored in a cell mapped to a variable in the storage. It is transformed into the following rewrite rule.

$$\begin{aligned} \mathcal{M}to\mathcal{R} \left(\frac{\mathbf{get_pre}(\iota, \mathit{sto})(l) = v}{l \xrightarrow{\iota} v} \right) = \\ \mathit{crl} : (l, \iota_{\mathbf{pre}}) \Rightarrow (v, \iota_{\mathbf{post}}) \text{ if } \iota_{\mathbf{pre}} = \iota_{\mathbf{post}} \wedge v = \mathbf{get_sto}(\iota_{\mathbf{pre}})(l). \end{aligned} \quad (17)$$

3.2 The Mapping from *ALTS* to \mathcal{R} -systems

The de-encapsulation of the information in a label in the mapping from *MSOS* to *RWL* is represented at the semantic level by the mapping $\mathcal{A}to\mathcal{T}\mathcal{S}$ from arrow-labeled transition systems to (unlabeled) transition systems. The label information is moved from the labels in the transitions of an *ALTS* \mathcal{A} to the configurations of the transitions of the transition system $\mathcal{T}\mathcal{S}(\mathcal{A})$, mapped from \mathcal{A} . The mapping $\mathcal{A}to\mathcal{T}\mathcal{S}$ accomplishes that by representing the label category $\mathbb{A}_{\mathcal{A}}$ as a *preorder*, $Pre(\mathbb{A})$, and then moving the elements of $Pre(\mathbb{A})$ into the configurations of $\mathcal{T}\mathcal{S}(\mathcal{A})$, defined as pairs composed of elements of the set of configurations of \mathcal{A} and $Pre(\mathbb{A})$. Note that the $Pre(\mathbb{A})$ have the same set *Index* as \mathbb{A} . Moreover, each element in $Pre(\mathbb{A})$ has the same indices as its counterpart in \mathbb{A} .

To describe how a label category \mathbb{A} is mapped into its preorder view $Pre(\mathbb{A})$ it is necessary to describe how values mapped to indices in a label $\alpha \in \mathbb{A}$ are mapped to values related to indices in elements of $Pre(\mathbb{A})$. The mapping of indices in the labels of $\mathbb{A}_{\mathcal{A}}$ built via the application of **ContextInfo** and **MutableInfo** label transformers is rather straightforward. For these two cases, the information mapped to an index i in a label α in \mathbb{A} is already present in the two components given by the application of the **pre** and **post** operations on α . That is, label categories built via the application of these two label transformers already have a preorder structure, with the ordering relation given by identity and pairing relations, respectively.

The main issue in the process of viewing \mathbb{A} as the preorder $Pre(\mathbb{A})$ relates to the treatment of indices built via the **EmittedInfo** label transformer. Such indices are mapped to free monoids. A free monoid is a special case of a preorder, when the prefix order of the free monoid is considered as the ordering relation. Nevertheless, in the context of arrow-labeled transition systems, to consider the prefix order of the monoid mapped to an **EmittedInfo**-declared index it is necessary to consider a “history” of emitted elements throughout a computation. This history is automatically made available if we consider label-sequences of derivatives in the *derivation tree* [10, Page 48] view of the transitions of \mathcal{A} , that represents branches or *paths* in the derivation tree of \mathcal{A} . Note that these paths actually represent computations in the transitions of \mathcal{A} . The label-sequence of a derivative captures exactly the notion of a “history” of labels from the initial state up to a certain (target) configuration in a computation.

Using lists as the data structure to model the history of values mapped to indices declared via an **EmittedInfo** label transformer, and the **pre** and **post** projection functions on labels for indices built using **ContextInfo** and **MutableInfo** label transformers, we define two functions, namely $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$, that compute the configurations of $\mathcal{T}\mathcal{S}(\mathcal{A})$ when applied to the derivatives of the derivation tree representation

of the transitions of \mathcal{A} . Note that the definitions for $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$ capture the notion of the preorder label category, since the second projection of the elements they return is of type $Pre(\mathbb{A})$.

To summarize, the mapping $\mathcal{A}to\mathcal{TS}$ is defined as in the context of an *MSOS* specification \mathcal{M} , generating a transition system $\mathcal{TS}(\mathcal{A})$ from an arrow-labeled transition system \mathcal{A} induced by \mathcal{M} such that the configurations of the transitions of $\mathcal{TS}(\mathcal{A})$ are given by the application of the functions $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$ to the derivatives of the paths in the derivation tree representation of the transitions of \mathcal{A} . The forthcoming definitions formalize this explanation.

Syntactical convention The expression $\alpha.i$ is an abbreviation for $get(\alpha, i)$.

Definition 5 (Preorder view of a Label Category). A preorder view of a label category \mathbb{A} , in the context of a *MSOS* specification \mathcal{M} , is represented as $Pre(\mathbb{A})$, such that for every label α in \mathbb{A} there exist two elements in $Pre(\mathbb{A})$, representing the label information prior, during, and posterior to the transition involving α .

The elements of $Pre(\mathbb{A})$ are tuples whose elements are the indices of the arrow structure of $Morph(\mathbb{A})$. The indices are mapped to values typed as follows.

- For an index i declared in $Lc_{\mathcal{M}}$ via a **ContextInfo**(i, T) declaration, $Pre(\mathbb{A}) = \langle T, id_T \rangle$, that is, the type of the values mapped to i in $Pre(\mathbb{A})$ is T with the ordering relation defined as the identity relation id_T .
- For an index i declared in $Lc_{\mathcal{M}}$ via a **MutableInfo**(i, T) declaration, $Pre(\mathbb{A}) = \langle T, \preceq \rangle$, that is, the type of the values mapped to i in $Pre(\mathbb{A})$ is T , with the ordering relation \preceq defined as follows. Given $a, b \in T$, $a \preceq b$ if $\langle a, b \rangle = \alpha.i$.
- For an index i declared in $Lc_{\mathcal{M}}$ via an **EmittedInfo**(i, T^*, \cdot, e), $Pre(\mathbb{A}) = \langle T^*, \leq \rangle$, that is, the type of the values mapped to i in $Pre(\mathbb{A})$ is $List[T^*]$, with the prefix ordering relation \leq according to [10].

Definition 6 (Path in a derivation tree). A path in a derivation tree is a branch of the tree beginning at the initial state and is represented as the set of derivatives that characterize that branch. The set of paths in the derivation tree of an *ALTS* \mathcal{A} is represented as $\Pi(\mathcal{A})$.

Definition 7 (Derivatives of an *ALTS*). The set of derivatives of an arrow-labeled transition system \mathcal{A} is defined as $\mathcal{D}(\mathcal{A}) = \{ \delta \mid \pi \in \Pi(\mathcal{A}), \delta \text{ is a derivative in } \pi \}$. The set of derivatives of a path π is defined as $\mathcal{D}_{\pi}(\mathcal{A}) = \{ \delta \mid \delta \text{ is a derivative in } \pi \}$.

Definition 8 (Partial functions $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$). Let us consider $\Pi(\mathcal{A})$, the set of paths of a derivation tree of an arrow labeled transition system \mathcal{A} induced by an *MSOS* specification \mathcal{M} ; γ_0 to γ_n configurations in $\Gamma_{\mathcal{M}}$; ι an identity label in $\mathbb{I}_{\mathbb{A}}$; $\alpha_1 \dots \alpha_n$ labels in $Morph(\mathbb{A}_{\mathcal{A}})$; ω and ω' elements of $Pre(\mathbb{A}_{\mathcal{A}})$; and δ a derivative in $\mathcal{D}(\mathcal{A})$, having the following form:

$$\delta = \{ (\iota, \gamma_0), (\alpha_1, \gamma_1), \dots, (\alpha_1 \dots \alpha_{n-1}, \gamma_{n-1}), (\alpha_1 \dots \alpha_{n-1} \alpha_n, \gamma_n) \}$$

Functions τ_1 and τ_2 are the first and second projection functions, respectively, for the pair $(\Gamma \times Pre(\mathbb{A}_{\mathcal{A}}))$.

The partial functions $preState_{\mathcal{A}}$ and $postState_{\mathcal{A}}$ are only defined for values $\pi \in \Pi(\mathcal{A})$ and $\delta \in \mathcal{D}(\mathcal{A})$ such that δ is a derivative in π . They have the following signatures.

$$\begin{aligned} preState_{\mathcal{A}} &: \Pi(\mathcal{A}) \times \mathcal{D}(\mathcal{A}) \rightarrow (\Gamma \times Pre(\mathbb{A})) \\ postState_{\mathcal{A}} &: \Pi(\mathcal{A}) \times \mathcal{D}(\mathcal{A}) \rightarrow (\Gamma \times Pre(\mathbb{A})) \end{aligned}$$

The function $preState_{\mathcal{A}}$ is defined inductively on the length of the derivatives of a path. The base case of $preState_{\mathcal{A}}$ is applied to a path π and the first immediate derivative in π . It is defined by the following equation,

$$preState_{\mathcal{A}}(\pi, (\alpha_1, \gamma_1)) = (\gamma_0, \omega) \quad (18)$$

where γ_0 is the initial configuration of \mathcal{A} and the indices of ω are associated to values in the following way:

- For an index i declared in $Lc_{\mathcal{M}}$ via a **ContextInfo**(i, T) label transformer, its value in ω is the same as in the label α_1 , that is, $\omega.i = \alpha_1.i = \mathbf{pre}(\alpha_1).i$.
- For an index i declared in $Lc_{\mathcal{M}}$ via a **MutableInfo**(i, T) label transformer, with v_1 and v_2 elements of type T , $\omega.i = v_1 = \mathbf{pre}(\alpha_1).i$ with $\alpha_1.i = (v_1, v_2)$.
- For an index i declared in $Lc_{\mathcal{M}}$ via an **EmittedInfo**(i, T^*, \cdot, e) label transformer, $\omega.i = \mathit{emptyList}$.

The inductive case of function $preState_{\mathcal{A}}$ is applied to a path π and a derivative in π , and is defined by the following equation,

$$preState_{\mathcal{A}}(\pi, (\alpha_1 \dots \alpha_{n-1} \alpha_n, \gamma_n)) = (\gamma_{n-1}, \omega) \quad (19)$$

where γ_{n-1} is such that $(\alpha_1 \dots \alpha_{n-1}, \gamma_{n-1})$ is a derivative in π , and the indices of ω are associated to values in the following way:

- For an index i declared in $Lc_{\mathcal{M}}$ via a **ContextInfo**(i, T) label transformer, $\omega.i = \alpha_n.i = \mathbf{pre}(\alpha_n).i$.
- For an index i declared in $Lc_{\mathcal{M}}$ via a **MutableInfo**(i, T) label transformer, with v_1 and v_2 elements of type T , $\omega.i = v_1 = \mathbf{pre}(\alpha_n).i$ with $\alpha_n.i = (v_1, v_2)$.
- For an index i declared in $Lc_{\mathcal{M}}$ via a **EmittedInfo**(i, T^*, \cdot, e) label transformer,

$$\omega.i = \mathit{append}(\alpha_{n-1}.i, (\tau_2(preState_{\mathcal{A}}(\pi, (\alpha_1 \dots \alpha_{n-1}, \gamma_{n-1}))))).i.$$

The function $postState_{\mathcal{A}}$ applied to π and a derivative in π , is defined by the following equation,

$$postState_{\mathcal{A}}(\pi, (\alpha_1 \dots \alpha_{n-1} \alpha_n, \gamma_n)) = (\gamma_n, \omega) \quad (20)$$

where the indices of ω' are associated to values in the following way:

- For an index i declared in $Lc_{\mathcal{M}}$ via a **ContextInfo**(i, T) label transformer, its value in ω is the same as in the label α_n , that is, $Pre(\mathbb{A}), \omega.i = \alpha_n.i = \mathbf{post}(\alpha_n).i$.
- For an index i declared in $Lc_{\mathcal{M}}$ via a **MutableInfo**(i, T) label transformer declaration, with v_1 and v_2 elements of type T , $\omega.i = v_2 = \mathbf{post}(\alpha_n).i$ with $\alpha_n.i = (v_1, v_2)$.

– For an index i declared in $Lc_{\mathcal{M}}$ via an **EmittedInfo** (i, T^*, \cdot, e) label transformer,

$$\omega.i = \text{append}(\alpha_n.i, (\tau_2(\text{preState}_{\mathcal{A}}(\pi, (\alpha_1 \dots \alpha_n, \gamma_n))))).i).$$

Definition 9 (The mapping from ALTS to TS). The mapping $AtoTS : \mathcal{A} \rightarrow \mathcal{TS}$, from ALTS to TS, when applied to an ALTS of the form $\mathcal{A} = \langle \Gamma, \text{Morph}(\mathbb{A}), \longrightarrow \rangle$, with Γ the configurations of \mathcal{A} , $\text{Morph}(\mathbb{A})$ the arrows of a label category, and \longrightarrow the transition relation declared as $\longrightarrow \subseteq \Gamma \times \text{Morph}(\mathbb{A}) \times \Gamma$, generates a transition system of the form $\mathcal{TS}(\mathcal{A}) = \langle (\Gamma \times \text{Pre}(\mathbb{A})), \longrightarrow_{\mathcal{TS}(\mathcal{A})} \rangle$, with the transition relation defined as

$$\longrightarrow_{\mathcal{TS}(\mathcal{A})} \subseteq (\Gamma \times \text{Pre}(\mathbb{A})) \times (\Gamma \times \text{Pre}(\mathbb{A})) \quad (21)$$

and $\text{Pre}(\mathbb{A})$ a preorder view of the label category \mathbb{A} . The elements of $(\Gamma \times \text{Pre}(\mathbb{A}))$ and $\longrightarrow_{\mathcal{TS}(\mathcal{A})}$ are given by the following equations, where $\gamma \in \Gamma$, π is a path in $\Pi(\mathcal{A})$ and δ a derivative in $\mathcal{D}_{\pi}(\mathcal{A})$.⁴

$$\mathcal{TS}(\mathcal{A}) = \langle \Gamma_{\mathcal{TS}(\mathcal{A})}, \longrightarrow_{\mathcal{TS}(\mathcal{A})} \rangle \quad (22)$$

$$\Gamma_{\mathcal{TS}(\mathcal{A})} = \{ \text{preState}_{\mathcal{A}}(\pi, \delta) \mid \pi \in \Pi(\mathcal{A}), \delta \in \mathcal{D}_{\pi}(\mathcal{A}) \} \cup \{ \text{postState}_{\mathcal{A}}(\pi, \delta) \mid \pi \in \Pi(\mathcal{A}), \delta \in \mathcal{D}_{\pi}(\mathcal{A}) \} \quad (23)$$

$$\longrightarrow_{\mathcal{TS}(\mathcal{A})} = \{ (\text{preState}_{\mathcal{A}}(\pi, \delta), \text{postState}_{\mathcal{A}}(\pi, \delta)) \mid \pi \in \Pi(\mathcal{A}), \delta \in \mathcal{D}_{\pi}(\mathcal{A}) \} \quad (24)$$

4 The Correctness Proof of $\mathcal{M}to\mathcal{R}$

The main idea behind the mapping from $MSOS$ to RWL is to “de-encapsulate” the information inside the labels in $MSOS$. According to Definition 4, this is accomplished syntactically by enriching the configurations of rewrite rules with label information. Semantically this means mapping the arrow-labeled transition system induced by an $MSOS$ specification to a(n) (unlabeled) transition system, as shown by Definition 9.

In order to prove that the mapping is correct, it is necessary to show that computations are preserved by the \mathcal{R} -system $T_{\mathcal{R}}$, the *initial* model of the rewrite theory generated from an $MSOS$ specification. The proof is twofold. Firstly, it is necessary to show that the transition system obtained from an arrow-labeled transition system by viewing its label category as a preorder preserves the computations of the original arrow-labeled transition system. Secondly, it should be proven that this transition system is the model for the rewrite theory resulting from the application of $\mathcal{M}to\mathcal{R}$ to that $MSOS$ specification.

The proof of Theorem 1 establishes that there exists a transition system associated with an arrow-labeled transition system induced by an $MSOS$ specification \mathcal{M} . Using the equivalence between \mathcal{R} -systems and transition systems originally presented in [9, Section 3.3], and summarized in Section 2.2, it is proven that there exists a \mathcal{R} -system equivalent to an arrow-labeled transition system. The proof of Theorem 2 shows that this \mathcal{R} -system is a model for the rewrite theory generated by the application of $\mathcal{M}to\mathcal{R}$ to \mathcal{M} .

⁴ Note that the partial functions $\text{preState}_{\mathcal{A}}$ and $\text{postState}_{\mathcal{A}}$ are applied to $\mathcal{D}_{\pi}(\mathcal{A})$ and not to $\mathcal{D}(\mathcal{A})$.

Theorem 1 (Preservation of computation in $\mathcal{A}to\mathcal{TS}$). *For every arrow-labeled transition system \mathcal{A} , in the context of specifications in MSOS, there exists a corresponding transition system $\mathcal{TS}(\mathcal{A})$ according to Definition 9 such that the states and transitions of \mathcal{A} are in a one-to-one correspondence to states and transitions of $\mathcal{TS}(\mathcal{A})$ and that all transitions of finite length of $\mathcal{TS}(\mathcal{A})$ are in \mathcal{A} .*

Proof Sketch. Since $\Pi(\mathcal{A})$ is the spanning tree of the graph view of the transition relation set of \mathcal{A} , having the initial state of \mathcal{A} as root, the elements of the transition relation set in $\mathcal{TS}(\mathcal{A})$ are in one-to-one correspondence to the elements of the transition relation set in \mathcal{A} , when the reflexive steps are removed and the transitive steps are expanded from $\mathcal{TS}(\mathcal{A})$.

The elements of the preorder $Pre(\mathbb{A})$ are tuples formed by values associated to the indices of the objects in the label category \mathbb{A} . Thus, the elements of the preorder $Pre(\mathbb{A})$ are in one-to-one correspondence to the indices of the objects in the label category \mathbb{A} . \square

Corollary 1 (ALTS— \mathcal{R} -system correspondence). *For every arrow-labeled transition system \mathcal{A} induced by a MSOS specification \mathcal{M} , there exists a corresponding \mathcal{R} -system for a rewrite theory \mathcal{R} that preserves the state information of \mathcal{A} , that preserves the computations of \mathcal{A} and that has its finite length transitions represented in \mathcal{A} .*

Proof Sketch. Applying the mapping from arrow-labeled transition systems to transition systems, to any ALTS \mathcal{A} , yields a corresponding transition system $\mathcal{TS}(\mathcal{A})$, according to Definition 9, that is a model \mathcal{S} of a rewrite theory \mathcal{R} via the equivalence between \mathcal{R} -systems and transition systems, defined in [9, Section 3.3]. \square

Theorem 2 shows that the rewrite theory obtained by the application of $\mathcal{M}to\mathcal{R}$ to \mathcal{M} is the very theory satisfied by \mathcal{S} mentioned in the proof sketch of Theorem 1. Note that \mathcal{S} is indeed the initial model of the rewrite theory $\mathcal{M}to\mathcal{R}(\mathcal{M})$.

Theorem 2 ($\mathcal{TS}(\mathcal{A})$ is the model for \mathcal{R}). *If \mathcal{M} is an MSOS specification, \mathcal{A} is the arrow-labeled transition system induced by \mathcal{M} , $\mathcal{TS}(\mathcal{A})$ is the transition system equivalent to \mathcal{A} and \mathcal{R} is the rewrite theory produced by the application of the mapping to \mathcal{M} then $\mathcal{TS}(\mathcal{A})$ is a model for \mathcal{R} .*

Proof Sketch. This proof is done by induction on the set of transition rules of \mathcal{M} and case analysis on the structure of each transition rule. It is shown that for every possible \mathcal{M} modeled by \mathcal{A} , $\mathcal{TS}(\mathcal{A})$ is a model for \mathcal{R} , and according to Corollary 1, $\mathcal{TS}(\mathcal{A})$ corresponds to a \mathcal{R} -system. \square

Corollary 2 (Correctness of the mapping). *Every transition t in an arrow-labeled transition system \mathcal{A} induced by a MSOS specification \mathcal{M} has a corresponding transition in \mathcal{S} , the \mathcal{R} -system that models the rewrite theory \mathcal{R} generated via the application of the mapping to \mathcal{M} .*

Proof Sketch. Corollary 1 proves that \mathcal{A} has an associated transition system $\mathcal{TS}(\mathcal{A})$, correspondent to a \mathcal{R} -system, that preserves its computations. Theorem 2 proves that $\mathcal{TS}(\mathcal{A})$ is a model for \mathcal{R} . \square

5 The MSOS-SL Interpreter

The MSOS-SL Interpreter [4, Chapter 6] is a prototype implementation of the $\mathcal{M}to\mathcal{R}$ mapping in the Maude system, a high-performance implementation of RWL [2]. The MSOS-SL Interpreter is built on top of Maude+rc, an extension of the Maude system that we have developed, that supports the so called *rewriting conditions*, that is, rewritings in the conditions of conditional rewrite rules. This extension is mainly a term evaluation strategy that implements a Prolog-like search. We refer to [4] for a complete explanation of Maude+rc.⁵

Formally, the mapping from $MSOS$ to RWL transformation process is a mapping $\phi_{MSOS} : \text{MsosModule} \rightarrow \text{EnStrOModule} \rightarrow \text{Module}$, where the sort `MsosModule` represents $MSOS$ specifications, the `EnStrOModule` represents object-oriented rewrite theories, and the sort `Module` represents rewrite theories.

The mapping from `EnStrOModule` \rightarrow `Module` was developed by Duran in his Ph.D. thesis [6] and is implemented as part of the Maude system.

The mapping from $MSOS$ to RWL is implemented in Maude as a further extension to the module algebra of Maude, built on top of Maude+rc. The module algebra extended by Maude+rc is further extended with the sort `MsosModule`, which defines a language that we have named MSOS-SL, an acronym for $MSOS$ specification language. Terms in `MsosModule` are transformed into terms in `EnStrOModule` following the transformations formally defined in Section 3, that is, the label category declaration in a term in `MsosModule` is transformed into a class declaration in a term in `EnStrOModule`, and the transition rule set declaration in a term in `MsosModule` is transformed into a rewrite rule set with rewrite conditions in `EnStrOModule`. This implementation defines the MSOS-SL Interpreter.

In [5] the present authors have proposed the Maude Action Tool (MAT), an executable environment for Action Semantics [11], a formalism for specifying the operational semantics of programming languages in a modular and readable way, using the reflective capabilities of Maude. The work presented in [5] was a prototype of MAT, in which we were experimenting with the ideas of the transformation from $MSOS$ to RWL , which are now formalized in Section 3. The rewrite theory representing the $MSOS$ specification of action notation, which was manually written for the MAT first prototype, is now automatically generated by the MSOS-SL Interpreter when the MSOS-SL specification of action notation is entered in the MSOS-SL Interpreter. Of course, the $MSOS$ specification of action notation is now one among a very wide variety of $MSOS$ specifications that the MSOS-SL Interpreter can execute. Due to space limitations, we can not present all the details of the relationship between MAT and the MSOS-SL Interpreter. For a complete presentation, we refer to [4, Chapter 6].

6 Conclusion

We have presented $\mathcal{M}to\mathcal{R}$, a proven correct mapping from modular structural operational semantics ($MSOS$) to rewriting logic (RWL). Such mapping allows the defini-

⁵ The *rcrl* rule extension to the Maude system that we have described in [5] evolved into Maude+rc.

tion of formal tools for *MSOS* such as the MSOS-SL Interpreter, our implementation for *MtoR* as a prototype in the Maude system. The MSOS-SL Interpreter allows the execution of *MSOS* specifications when written using the MSOS-SL specification language: a Maude-like language with support for the declaration of label categories and (arrow-)labeled transition rules.

Another possible mapping from *MSOS* to *RWL* would consider labeled transitions as terms in the generated rewrite theory. This is the so-called sequent-based mapping [7]. This approach leads indeed to a simpler translation scheme. However our mapping derives a direct interpreter for *MSOS*. The former mapping would require existential queries in order to execute a *MSOS* specification in a prolog-like way. From an efficiency point of view and in the context of a Maude implementation both approaches are comparable assuming that rewriting conditions are implemented in the Maude virtual machine. In [13] a *MSOS* prolog-based executable environment following the sequent-based approach is presented. The efficiency of MSOS-SL Interpreter will only be reasonable and thus comparable to such implementations or usable by third parties when rewriting conditions are native implemented in the Maude virtual machine. This feature will be available in the next version of the Maude system, Maude 2.0. Another interesting issue regarding MSOS-SL Interpreter when compared to a prolog-based implementation is the possibility of incorporating the model-checking facilities that the Maude 2.0 will provide.

Our future work will focus on two main subjects discussed above: improving the efficiency of the MSOS-SL Interpreter and adding model checking capabilities to the MSOS-SL Interpreter, evolving the interpreter into a tool set. At the implementation level, these two tasks are closely coupled with the next release of the Maude 2.0 system, since rewriting conditions are scheduled to be supported and a model checker for linear time temporal logic is already present in the most recent alpha releases of the Maude system.

References

1. M. Clavel. *Reflection in rewriting logic: metalogical foundations and metaprogramming applications*. CSLI Publications, 2000.
2. M. Clavel, F. Durán, S. Eker, N. Martí-Oliet, P. Lincoln, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, <http://maude.csl.sri.com>, January 1999.
3. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, August 2002.
4. C. de O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontificia Universidade Católica do Rio de Janeiro, September 2001. <http://www.ic.uff.br/~cbraga>.
5. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *AMAST'00, Proc. 8th Intl. Conf. on Algebraic Methodology and Software Technology, Iowa City, IA, USA*, volume 1816 of *Lecture Notes in Computer Science*, pages 407–421. Springer, May 2000.
6. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Mlaga, Escuela Tcnica Superior de Ingeniera Informtica, 1999.

7. N. Martí-Oliet and J. Meseguer. *Handbook of Philosophical Logic*, volume 61, chapter Rewriting Logic as a Logical and Semantic Framework. Kluwer Academic Publishers, second edition, 2001. <http://maude.csl.sri.com/papers>.
8. J. Meseguer. A logical theory of concurrent objects. In N. Meyrowitz, editor, *Proceedings ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM Press, 1990.
9. J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.
10. R. Milner. *Communication and Concurrency*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1989.
11. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
12. P. D. Mosses. A Modular SOS for ML concurrency primitives. Technical Report Research Series RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.
13. P. D. Mosses. Pragmatics of Modular SOS. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.
14. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN - 19, Computer Science Department, Aarhus University, 1981.