Implementing Modular SOS in Maude

Fabricio Chalub and Christiano Braga

Universidade Federal Fluminense, Niterói, Brazil

Abstract. Modularity is a pragmatic property of specifications that is not easy to achieve. For instance, it has been left as an open problem by Plotkin in his 81 Aarhus lecture notes where Structural Operational Semantics (SOS) was defined. This open problem has been solved only recently by Mosses with Modular SOS (MSOS), a framework that extends labelled transitions systems with a label category where the semantic information is *encapsulated* inside its arrows. This extension gave rise to *arrow-labelled transition systems* that allow MSOS specifications to be made modular, that is, extended monotonically.
The objective of this paper is to present the Maude MSOS Tool, a Maude implementation of MSOS. Maude is a fast implementation of rewriting logic, a *reflective* logic that has been shown as a generic framework which can represent many logics, specification languages and models of computation. It is precisely the reflective capabilities of rewriting logic implemented in the Maude system that allow us to create an executable environment for MSOS: Maude MSOS Tool.

1 Introduction

Structural operational semantics (SOS) is a simple yet mathematically rigorous generic semantic framework. Perhaps due to these two properties, is widely used in a number of different classes of problems, such as language semantics [19] and formal models of concurrency [18]. However, despite its wide application, an important pragmatic property is missing in SOS: modularity, a key feature for the engineering of large specifications. Roughly, it means that if one should make an extension to an existing specification to add new features that would involve the addition of a new semantic component, this would imply a change to all existing rules. That is, specifications in SOS are *non-monotonic*.

Recently Peter Mosses has solved the modularity problem in SOS with a new framework called Modular SOS (MSOS) [23]. In a nutshell, Mosses uses the labels in the transitions to encapsulate the semantic information as records that may grow as new language features are added in an extension to an existing MSOS specification.

Meseguer and Braga have developed a mapping [16] from MSOS to rewriting logic [13], a general *reflective* formalism that may represent several logics, specification languages and models of computation. This mapping allows for the application of the many execution, analysis and verification tools available for rewriting logic implementations, in particular those developed in Maude [6], including a LTL model checker [9] and Full Maude [8], an extensible module algebra for the Maude language.

The goals of this paper are to explain how we have used the reflective capabilities of the Maude system and Full Maude to implement the Maude MSOS Tool, an

executable environment for MSOS specifications, based on the formally defined mapping from MSOS to rewriting logic, and to show how one may execute and verify programs, written in a language $\mathcal{L}$, based on $\mathcal{L}$'s MSOS semantics using the Maude System. It is worth mentioning that the Maude MSOS Tool is available for download from `http://www.ic.uff.br/~cbraga/losd/maude-msos-tool/`.

To achieve these goals we begin our paper motivating the MSOS framework with an exposition of the modularity problem in SOS. This is accomplished in Section 2. Section 3 gives some background in rewriting logic and presents the mapping from MSOS to rewriting logic. In Section 4 we present MSOS-SL, the specification language available in the Maude MSOS Tool, defined as a conservative extension of the Maude language. In Section 5 we present how we have extended Full Maude to create the Maude MSOS Tool, using Maude's reflective capabilities. Section 6 gives two examples of how to execute a specification in the Maude MSOS Tool. The first example shows how a simple functional *sequential* program may be executed and the second example shows how a *concurrent* functional program may be executed, exploring Maude's logic programming capabilities. We conclude this paper in Section 7 with our final remarks.

2 Modularity in Structural Operational Semantics Specifications

The Structural Operational Semantics (SOS) framework, defined by Gordon Plotkin in [24], is a commonly used framework for the definition of formal programming languages semantics. Plotkin left open the problem of modularity in SOS specifications. For example, Rules 1 and 2 define the SOS, also called "small-step" operational semantics, for simple mathematical expressions:

$$Exp ::= m \mid e_0 \texttt{ + } e_1 \qquad m \in \mathbb{N}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0 \texttt{ + } e_1 \longrightarrow e_0' \texttt{ + } e_1} \qquad \frac{e_1 \longrightarrow e_1'}{m_0 \texttt{ + } e_1 \longrightarrow m_0 \texttt{ + } e_1'} \tag{1}$$

$$m_0 \texttt{ + } m_1 \longrightarrow m_0 + m_1 \tag{2}$$

The addition of bindings requires the use of an *environment* component, added to the configuration. We use here an example adapted from Plotkin's notes, which is a simpler form of the **let** construct from Standard ML [19]. Rule 3 specifies that, first, the expression e_0 is evaluated until a final value m is found. Rule 4 specifies that e_1 should be evaluated into e_1' in the context of a new environment, obtained by replacing all instances of the variable x in the environment ρ by m and placing back the evaluated e_1' into the **let** body. Finally, Rule 5 specifies that when e_1 is evaluated to a final value n, the entire expression should be replaced by n.

$$Exp ::= \texttt{ let } x \texttt{ = } e_0 \texttt{ in } e_1 \texttt{ end} \qquad x \in \mathrm{Var} = \{x_1, x_2, \ldots\}$$

$$\frac{\rho \vdash_V e_0 \longrightarrow e_0'}{\rho \vdash_V \texttt{ let } x \texttt{ = } e_0 \texttt{ in } e_1 \texttt{ end} \longrightarrow \texttt{ let } x \texttt{ = } e_0' \texttt{ in } e_1 \texttt{ end}} \tag{3}$$

$$\frac{\rho[m/x] \vdash_{V \cup \{x\}} e_1 \longrightarrow e_1'}{\rho \vdash_V \texttt{ let } x \texttt{ = } m \texttt{ in } e_1 \texttt{ end} \longrightarrow \texttt{ let } x \texttt{ = } m \texttt{ in } e_1' \texttt{ end}} \tag{4}$$

$$\rho \vdash_V \texttt{ let } x \texttt{ = } m \texttt{ in } n \texttt{ end} \longrightarrow n \tag{5}$$

Since now expressions are evaluated in the presence of an environment, the rules for mathematical expressions must be rewritten.

$$\frac{\rho \vdash_V e_0 \longrightarrow e_0'}{\rho \vdash_V e_0 \texttt{ + } e_1 \longrightarrow e_0' \texttt{ + } e_1} \qquad \frac{\rho \vdash_V e_1 \longrightarrow e_1'}{\rho \vdash_V m_0 \texttt{ + } e_1 \longrightarrow m_0 \texttt{ + } e_1'} \tag{6}$$

$$\rho \vdash_V m_0 \texttt{ + } m_1 \longrightarrow m_0 + m_1 \tag{7}$$

To solve the modularity problem, Peter Mosses developed a framework called Modular Structural Operational Semantics (MSOS) [23].

The key modularity point in MSOS lies on the *arrow-labelled transition systems* where the semantic components such as the environment are moved from the configurations to the transition label, now structured as a record, and referred through *indices*. Transitions in an arrow-labelled transition system are understood as arrows in a category, named label category. The configurations rewritten by the transition rules consist only of value-added abstract syntax trees, that is, an abstract syntax tree that may have computed values on its branches. The idea is that a label may contain an unspecified number of components, but *only the components that are needed* in a particular transition must be made explicit.

Let us illustrate the modularity of MSOS specifications revisiting the specifications for arithmetic expressions and let expressions. Rules 8 and 9 specify the evaluation of expressions in MSOS. An informal explanation for Rule 8 is as follows: to evaluate e_0 + e_1, first evaluate one step of e_0, giving e_0'. The label X in Rule 8 means that while evaluating expression e_0, any changes to the semantic components (such as a memory component, but not an environment) should be part of the transition in the rule's conclusion. The label may, of course, remain the same. This is the case of Rule 9, where label U means that no change may happen at all in the semantic components, that is, the label is an *identity* label with respect to label composition. For a label to be *unobservable* it must have: (i) $X.i = X.i'$, in usual record notation, for a read-write index i; (ii) $X.i' = \varepsilon$, for write-only index i.

$$\frac{e_0 \xrightarrow{X} e_0'}{e_0 \texttt{ + } e_1 \xrightarrow{X} e_0' \texttt{ + } e_1} \qquad \frac{e_1 \xrightarrow{X} e_1'}{m_0 \texttt{ + } e_1 \xrightarrow{X} m_0 \texttt{ + } e_1'} \tag{8}$$

$$m_0 \texttt{ + } m_1 \xrightarrow{U} m_0 + m_1 \tag{9}$$

To give semantics to a **let** expression, we should now add an environment to the specification by means of an index declaration in the labels. Indices of labels may be of three different types, that reflect on the different types of components they refer to: (i) read-only, declared with an un-primed index. This index represents the information that may not change, typically represented as an environment component; (ii) read-write, that actually declares a *pair* of indices, an un-primed and a primed, typically

represented as a memory component. The un-primed index represents the information available in the primed index of an adjacent transition, and the primed index represents the component that may have changed during the transition itself; and (iii) write-only, declared as a single, primed, index. In this case, the primed index represents the information emitted by (or resulting from) the transition. Write-only components from a set S form a monoid $(S^*, \cdot, \varepsilon)$, where $\cdot$ is the monoid binary operation, and ε is the monoid identity element.

Rules 10, 11, and 12 specify in MSOS the meaning of let expressions. The informal description of Rule 11 is: to evaluate the e_1 expression inside the **let**, evaluate one step of e_1 in the context of a new env-indexed component ($\rho[m/x]$) into e_1'; any changes to unspecified components (represented by the notation "...") should be carried onto the main rule. We have omitted here, and from the SOS specification above, the rules for the evaluation of variables.

$$\frac{e_0 \ -X\rightarrow \ e_0'}{\texttt{let}\ x\ \texttt{=}\ e_0\ \texttt{in}\ e_1\ \texttt{end}\ \ -X\rightarrow\ \ \texttt{let}\ x\ \texttt{=}\ e_0'\ \texttt{in}\ e_1\ \texttt{end}} \tag{10}$$

$$\frac{e_1 \ -\{(env : \rho[m/x]), \ldots\}\rightarrow \ e_1'}{\texttt{let}\ x\ \texttt{=}\ m\ \texttt{in}\ e_1\ \texttt{end}\ \ -\{(env : \rho), \ldots\}\rightarrow\ \ \texttt{let}\ x\ \texttt{=}\ m\ \texttt{in}\ e_1'\ \texttt{end}} \tag{11}$$

$$\texttt{let}\ x\ \texttt{=}\ m\ \texttt{in}\ n\ \texttt{end}\ \ -U\rightarrow\ n \tag{12}$$

Finally, computations are sequences of labelled transitions between configurations, with the additional requirement that the labels of adjacent transitions are composable. The composition of two labels $X = X_1; X_2$ is defined as: (i) for each read-only index i, $X.i = X_1.i = X_2.i$; (ii) for each read-write index i, $X.i = X_1.i$ and $X.i' = X_2.i'$; (iii) for each write-only index i, $X.i' = (X_1.i' \cdot X_2.i')$.

3 Mapping MSOS to Rewriting Logic

This section aims to present the formal mapping from MSOS to rewriting logic. This mapping was given by Meseguer and Braga in [16] with a detailed presentation. Here we will present the intuition and main points of the mapping and its correctness proof, necessary for a complete description of the Maude MSOS Tool as a formally defined and implemented meta-tool in rewriting logic. We refer the interested reader to [16] for the detailed presentation.

Let us begin with a short introduction to rewrite theories in rewriting logic.

A rewrite theory in rewriting logic [13] is a triple (Σ, E, R) with Σ the signature of the rewrite theory, E the set of Church-Rosser equations, and R the set of weakly coherent rewrite rules that are applied modulo the equations. The rewriting logic calculus is given by the rules of deduction[1] in Figure 1.

[1] Recently, Bruni and Meseguer formalized a *generalized* version of rewriting logic [4], in which *frozen arguments* may be specified on the operations. A *frozen argument* allows no rewrites under it. Since we do not explore this feature in our mapping, the rules of deduction in Figure 1 do not mention this new feature.

$$\frac{t \in \mathbb{T}_\Sigma(X)_k}{(\forall X)\ t \to t}\ \textbf{Reflexivity} \qquad \frac{(\forall X)\ t_1 \to t_2, \qquad (\forall X)\ t_2 \to t_3}{(\forall X)\ t_1 \to t_3}\ \textbf{Transitivity}$$

$$\frac{E \vdash (\forall X)\ t = u, \qquad (\forall X)\ u \to u', \qquad E \vdash (\forall X)\ u' = t'}{(\forall X)\ t \to t'}\ \textbf{Equality}$$

$$\frac{\begin{array}{c} f \in \Sigma_{k_1 \cdots k_n, k}, \qquad t_i, t_i' \in \mathbb{T}_\Sigma(X)_{k_i} \ \text{ for } \ i \in \{1, \ldots, n\} \\ (\forall X)\ t_j \to t_j' \ \text{ for } \ j \in \nu(f) \end{array}}{(\forall X)\ f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')}\ \textbf{Congruence}$$

$$\frac{\begin{array}{c} (\forall X)\ r \colon t \to t' \text{ if } \bigwedge_{i \in I} p_i = q_i \ \wedge \ \bigwedge_{j \in J} w_j : s_j \ \wedge \ \bigwedge_{l \in L} t_l \to t_l' \in R \\ \theta, \theta' \colon X \to \mathbb{T}_\Sigma(Y), \\ E \vdash (\forall Y)\ \theta(p_i) = \theta(q_i) \ \text{ for } \ i \in I, \qquad E \vdash (\forall Y)\ \theta(w_j) : s_j \ \text{ for } \ j \in J \\ (\forall Y)\ \theta(t_l) \to \theta(t_l') \ \text{ for } \ l \in L, \qquad (\forall Y)\ \theta(x) \to \theta'(x) \ \text{ for } \ x \in \nu(t, t') \end{array}}{(\forall Y)\ \theta(t) \to \theta'(t')}\ \begin{array}{l}\textbf{Nested} \\ \textbf{Replacement}\end{array}$$

Fig. 1. Deduction rules for rewriting logic.

Rewriting logic is a *computational logic* for the specification of concurrent systems [13]. Its inference system allows us to infer all the possible finitary concurrent computations of a system specified as a rewrite theory. One could then make a computational reading of the rules of deduction in Figure 1 as follows: (i) reflexivity is just the possibility of having idle transitions, (ii) equality means that states are equal *modulo* the set of equations E, (iii) congruence is a general form of *sideways* parallelism, (iii) replacement combines an atomic transition at the top using a rule with nested concurrency in the substitution, and (iv) transitivity is sequential composition.

Let us turn now to the use of rewriting logic in the context of giving *modular* semantics specifications to programming languages.

Modularity of language semantics specifications can be achieved in rewriting logic directly using a technique named modular rewriting semantics (MRS) [16] which essentially defines a *special form* of a rewrite theory designed specifically for this purpose.

MRS specifications use a syntax-directed style of semantics, with program syntax being separated from semantic components, such as the environment, memory or synchronization signals. This is captured by the operator `<_,_> : Program Record -> Conf`, a constructor of sort `Conf`. The sort `Program` represents a program's syntax and the sort `Record` is precisely the sort that holds the semantic components together by means of indices. The association between an index and a semantic component, or component for short, is called a field, declared with syntax `_:_ : [Index] [Component] -> [Field]`. A subset of a record is called a pre-record. A record then is a multiset of non-duplicated pre-record fields declared with the operator `{_} : [PreRecord] -> [Record]`. The `RECORD` theory in Maude is given below.

```
fmod RECORD is
```

```
 sorts Index Component Field PreRecord Record Truth .
 subsort Field < PreRecord .
 op tt : -> Truth .
 op null : -> PreRecord [ctor] .
 op _,_ : PreRecord PreRecord -> PreRecord
                                 [ctor assoc comm id: null] .
 op _:_ : [Index] [Component] -> [Field] [ctor] .
 op {_} : [PreRecord] -> [Record] [ctor] .
 op duplicated : [PreRecord] -> [Truth] .
 var I : Index . vars C C' : Component . var PR : PreRecord .
 eq duplicated((I : C),(I : C'), PR) = tt .
 cmb {PR} : Record if duplicated(PR) =/= tt .
endfm
```

When a sort is used between angle brackets in a operation declaration, such as `[Field]` in the declaration of the operator `_:_`, it means that the function represented by that operator is *partial*. An operator declaration may also have attributes, such as `assoc`, meaning that the operator is associative, `comm`, meaning commutativity and `ctor`, meaning that the operator is a constructor. The operator `_:_` for instance is a constructor for terms of sort `Field`.

This theory, of course, is not enough to guarantee the modularity of a language semantics specification. Two techniques should be used to make sure that once a rewrite rule is written to give semantics for a language feature, that is, when a semantic rule is specified, it is done once and for all.

The first technique is called *record inheritance* and consists of adding a new field to the record structure whenever a new semantic component is necessary. This technique enforces modularity when the semantic rules only make explicit the fields necessary to specify a certain language feature, and are, therefore, definitive in the precise sense advocated by Mosses in his Definitive Semantics notes [22].

This technique may be illustrated with the example from Section 2. An environment would be declared as a component bound to an index `env` giving rise to the field:

```
op env : -> Index .
mb env : E:Env : Field .
```

The membership declaration, done with syntax `mb` specifies that when the index `env` is related to and environment `E:Env` by the operator `:`, a field is formed. (Membership equational logic [14] is a generalization of order-sorted equational logic.) If this example specification were to be further extended with a store component, a similar declaration would be then necessary but this time declaring and index, let us say, `sto` and a membership equation binding `sto` to a term of sort `Store`, the data type for memory stores.

The semantic rules for the **let** expressions and memory assignment, for instance, are definitive. Neither the former would need to be changed when stores are added nor the latter in a further extension such as the inclusion of concurrency primitives.

The second technique is the systematic use of *abstract functions*, or interfaces, in the specification of semantic rules while referring to the application of operations to

semantic components. If such functions were made concrete, that is, equationally axiomatized at the level of semantic rules, a given semantic rule would be committed to a particular implementation of a data type. This is problematic when one wishes to implement analysis tools based on programming languages semantics specifications, such as in [3], which may require several extensions to concrete data type structures.

To exemplify the use of abstract functions, let us consider a scenario where one needs to commit to a particular garbage collection strategy. This would require a fine tuning on the semantics of the declaration of references and memory cell allocation. If Rule 11 referred to a concrete overriding function, with syntax `/` in Rule 11, the new requirement of garbage collection would imply a *retract* of Rule 11 and the addition of a new rule. Since abstract functions are being used, one only needs to extend the concrete implementation of environment (and stores) with the new functionality.

The question now is how to make use of these techniques to relate MSOS to rewriting logic.

First of all, we need to extend the `RECORD` theory worth sorts `IRecord`, for identity records, subsort of `Record`, `IPreRecord`, for identity prerecords, subsort of `PreRecord`, and `ROPreRecord`, `WOPreRecord`, and `RWPreRecord`, for multisets of fields that declare indices that should be read-only, write-only, and read-write, respectively, all subsorts of `PreRecord`. A `WOPreRecord` associates an index with a *free monoid* component, and declares a prefix predicate $\sqsubseteq$, where $C \sqsubseteq C'$ means that for each write-only field k the string $C.k$ is a possibly identical prefix of the string $C'.k$. Composition of records is also defined, with syntax `;`, such that for the composition $u; u'$ to be defined, we must have $u.i = u'.i$ for each read-only index i, and $u.j' = u'.j$ for each read-write index j. The composition $u; u'$ then has $(u; u').i = u.i = u'.i$, $(u; u').j = u.j$, $(u; u').j' = u'.j'$, and $(u; u').k = (u.k).(u'.k)$ for each write-only index k. *Identities* are then records u such that $u.j = u.j'$, and $u.k = nil$.

With this extended record theory we can now represent MSOS labeled transitions $t \xrightarrow{u} t'$ as transitions with t and t' being program expressions an u being a *record*. Therefore the modularity of MSOS inference rules is precisely captured by the *record inheritance* technique.

Let us consider the following general form of a MSOS inference rule defining the semantics of a language feature f,

$$\frac{v_1 \xrightarrow{u_1} v_1' \ \ldots \ v_n \xrightarrow{u_n} v_n' \quad cnd}{f(t_1, \ldots, t_n) \xrightarrow{u} t'} \tag{13}$$

where $f(t_1, \ldots, t_n), t'$, and the v_i, v_i' are program expressions (which can involve values), $u, u_1, \ldots, u_n$ are record expressions, and cnd is a side condition involving equations and perhaps predicates.

Our mapping is based on the notation given in [23], with the extra requirements that: (i) the side conditions cnd do not involve any expressions with records, fields, indices or `[Truth]` values, and (ii) records are expressed either in its explicit form using the $\{\_\}$ constructor or as a variable of sort `Record`.

With the representation of labels as records and the extra requirements on the use of record expressions in the transition rules, once we choose membership equational logic to specify the syntax and the semantic components the equational part of the resulting

rewrite theory is pretty much defined, and we need now to specify how the inference rules are translated to conditional rewrite rules. This is done following the intuition of building a *preorder* out of the transition relation of the arrow-labelled transition system associated with MSOS specifications. This intuition gives rise to the following translation from records (that is, labels) in a MSOS inference rule to record *projections* in conditional rewrite rule. Given a record u in a MSOS transition u is deconstructed into its *pre* and *post* projections that represent the semantic components before and after a transition involving u. The u^{pre} and u^{post} projections are calculated as follows.

- If $i : w$ is a `ROPreRecord` it appears as is in both u^{pre} and u^{post} projections.
- If $i' : w$ is a `WOPreRecord`, then the u^{pre} projection has $i : l$, with l a variable of sort S, with $(S^*, \cdot, \varepsilon)$ the free monoid related to i, and u^{post} has $i : l \cdot w$. If $i' : w$ appears on a transition in the *condition*, then the u^{pre} projection has $i : \varepsilon$ and the u^{post} has $i : w$.
- If $i : w$ and $i' : w'$ are `ROPreRecord` field expressions in u, then $i : w \in u^{pre}$ and $i' : w' \in u^{post}$.

Given a MSOS specification $\mathcal{S}$, this transformation is semantics-preserving in the precise sense of taking the form of a strong bissimulation of the labeled transition systems associated with the initial reachability preorder restricted to sort `Conf` that models the rewrite theory generated from $\mathcal{S}$ and the category of finite computations defined by $\mathcal{S}$. Again, we refer the interested reader to [16, 15] for a detailed presentation and a proof sketch of the bissimulation, respectively.

4 MSOS-SL

MSOS-SL is a specification language for MSOS, defined as a conservative extension of Full Maude's system modules and therefore follow the algebraic way of specifying language semantics.

MSOS-SL modules are declared with syntax `(msos` $\langle name \rangle$ `is` $\langle includes \rangle \ldots$ `sosm)`. The $\langle includes \rangle$ part of the module definition is a sequence of `including` $\langle m \rangle$ `.` declarations. MSOS-SL modules contain three distinct parts: the signature of the object language, that is, the language being specified, the declaration of the indices in the labels, and the declaration of transitions.

In what follows, we will describe the syntax of each part of MSOS-SL specifications and show how the example on Section 2 is specified in MSOS-SL.

4.1 Defining the signature of the object language

The formal definition of the syntax of a programming language is usually stated as productions in the Backus-Naur Form (BNF), as we have done in Section 2. Maude's *functional modules*, with membership equational logic (**mel**) as its underlying equational logic, offer a flexible alternative: operations may be defined in *infix* notation, have associated precedence, gathering patterns, and attributes such as associativity, commutativity, and identity (left, right, or both).

In what follows, we introduce the main constructions for the syntax definition of programming languages by making a parallel with BNF declarations. For this simple example it suffices to say that nonterminals in BNF productions are converted into *sort declarations* in Maude and, for a production rule such as $P \rightarrow \gamma$ we have: i) if γ is a non-terminal, the production is mapped into the declaration of the sort corresponding to γ as a subsort of the sort corresponding to P; ii) if γ is a combination of terminals and non-terminals, it is mapped into an operator declaration in Maude. For a more thorough discussion on the mapping between context-free grammars and a simpler version of **mel**, order-sorted equational logic, see [10].

Care must be taken when the production γ is a terminal symbol, but belongs to an *infinite set* of constants, such as $\mathbb{N}$. In Maude, the infinitely enumerable constants in such sets must, of course, be represented by an algebraic data type. The set of integers might be specified, for example, in the traditional Peano notation. Thus, the sort of the algebraic data type related to γ should, in this particular case, be a subsort of the sort corresponding to P.

For example, the BNF rules for the declaration of expressions, as defined in Section 2 is:

$$Exp ::= m \;\mid\; e_0 \texttt{ + } e_1 \qquad m \in \mathbb{N}$$

Assuming that we have a predefined sort `Int` that represents the integers, the corresponding translation to MSOS-SL is the declaration of the sort `Exp`, its supersort relation with `Int`, and the declaration of the `_+_` operator. As mentioned before, the declaration of an operator in Maude is made using the `op` keyword. The `ctor` operator attribute specifies that this is a *constructor* operator, and not a function over terms. When an operator has an underline ("_"), it is said to be in *infix form* and arguments will occupy the places of the underline character in terms. Thus in Maude this all takes the following form.

```
sort Exp .
subsort Int < Exp .

op _+_ : Exp Exp -> Exp [ctor] .
```

Finally, the BNF rule for **let** expressions is:

$$Exp ::= \texttt{let } x \texttt{ = } e_0 \texttt{ in } e_1 \texttt{ end} \qquad x \in \mathrm{Var} = \{x_1, x_2, \ldots\}$$

And its corresponding MSOS-SL fragment is the declaration of the sort `Var` and the declaration of the `let_in_end` operator, as follows:

```
sort Var .
op let_=_in_end : Var Exp Exp -> Exp [ctor] .
```

4.2 Declaring label indices

Label indices in MSOS-SL may be declared using the following keywords:

```
read-only i : τ .
read-write i : τ .
write-only i : τ (e, bop) .
```

where i is the index name, and τ the sort of the values indexed by i, referred to as *components*.

The declaration of read-write indices creates two indices, i and i', as outlined in Section 2. The declaration of the write-only index i creates one index i'. Also, this declaration needs the information about the corresponding monoid component: its identity element ($\mathbf{e}$) and the binary operation ($\mathbf{bop}$).

A component is specified as an algebraic data type in a Maude functional module. For example, we may define the environment for bindings by declaring a sort `Env`, for the component itself, and a sort `BVal` which represents all values that may be bound to an identifier, and associated functions, such as disjoint union of bindings, overriding of one set of bindings by another, a bind constructor that creates bindings from identifiers and `BVal` terms, and a lookup function that given an environment and an identifier, returns the value bound to it in the given environment. The module `ENVIRONMENT` declares the signature of the environment data type declaring operator `_U_` as the disjoin union of bindings, `_->_` as the bind constructor, `_//_` as the overriding function, and `find` as the lookup function.

```
(fmod ENVIRONMENT is protecting IDE .
 sorts Env BVal .

 op _U_ : Env Env -> Env .       op find : Env Ide -> [BVal] .
 op _->_ : Ide BVal -> Env [ctor] . op _//_ : Env Env -> Env .
 ...
endfm)
```

The use of `[BVal]` as the image sort for the function `find` means that this represents a *partial* function that might return an *error term* at the kind level. (A kind [14] is the connected component of sorts related by the subsort relation). For example, when the identifier is not found on the given environment, the function should return an error term, such as `no-value`.

It is important to emphasize that functional modules in Maude are expected to be confluent and terminating that is, Church-Rosser. Therefore the theories that specify the data types for the indices values are supposed to have the Church-Rosser property.

Returning to our example in Section 2, the declaration of the index env as the index for the environment component used in Rules 10 and 11 is specified in MSOS-SL using a read-only index declaration as follows:

```
read-only env : Env .
```

4.3 Declaring transition rules and label expressions

Transitions in MSOS-SL are declared with syntax `ctr` in the following way:

`ctr` γ `=` α `=>` γ' `if` $\langle condition \rangle$ `.`

where α has sort `Label`. Labels are formed by a set of *fields* of the form `(`i `:` C`)`, where i is a previously declared index term and C its associated component term. The set of fields is separated by commas and enclosed by braces (see example in the next paragraph). Also, γ is the value-added abstract syntactic tree before the execution of the rule and γ' is the result of the execution of the rule.

The sort `IndexSet` is defined as a subsort of a `Label`. This opens the possibility to create *label expressions* as in MSOS. In the expression `{(env : rho), (st : sigma), (st' : sigma'), IS}`, the variable `IS`, of sort `IndexSet`, matches against any unspecified component.

Unobservable labels are *identity labels* of the sort `ILabel`, a subsort of `Label`, and their subsets are of the sort `IIndexSet`, a subsort of `IndexSet`.

The $\langle condition \rangle$ part of MSOS-SL transitions consists of a conjunction of transitions, with syntax "`/\`" written in the general form γ `=` α `=>` γ', together with the usual conditions from Maude system modules:

- ordinary equations `t = t'`, which are satisfied if and only if the canonical forms of `t` and `t'` are equal modulo the equational attributes specified in the operators in `t` and `t'`, such as associativity, commutativity, and identify.
- abbreviated Boolean equations such as `t`, abbreviating the equation `t = true`. There are a number of built-in predicates, such as: equality (`_==_`), inequality (`_=/=_`), membership predicates (`_:: S`, with `S` a sort, which returns true if the parameter is of sort `S`), together with a combination of the connectives `not_`, `_and_`, and `_or_`.
- matching equations [6], written as `t := t'`, which are also ordinary equations, but with additional requirements at the operational level. In essence, matching equations are used to instantiate new variables by matching the lefthand side of the matching equation against the righthand side.
- rewrites, such as `t => t'`, where `t` and `t'` are terms of any sort, which means that there is a rewrite of the term `t` to the term `t'` with zero or more rewriting steps.

If the transition is unconditional it can be written simply as `tr` γ `=` α `=>` γ' `.` Finally, unobservable transitions have the alternative syntax γ `==>` γ'.

Let us now exemplify the declaration of transitions for **let** expressions whose meaning was given by the rules give in Section 2. The **let** rules may be written in MSOS-SL as follows. For brevity, we have omitted the rules that govern the evaluation of declarations, such as `val x = (1 + 10) and y = 5`, which, when evaluated, generate a set of bindings.

```
var X : Label .  var  IS : IndexSet .
var v : Value .  vars D D' : Decl . var i : Var.
vars E1 E'1 E2 E'2 : Exp . vars b rho rho' : Env .

ctr let x = E1 in E2 end = X => let x = E1 in E2 end
if E1 = X => E'1 .

ctr let x = v in E2 end ={(env : rho), IS}=>
    let x = v in E'2 end
```

```
if rho' := (rho // (x->v)) /\ E2 ={(env : rho'), IS}=> E'2 .

tr let x = b in v end ==> v .
```

The MSOS-SL transition rules above specify the inference Rules 10, 11, and 12 on page 4, respectively.

5 Maude MSOS Tool as a Formal Meta-Tool in Maude

Rewriting logic is reflective in the strict sense in which there exists an universal rewrite theory $\mathcal{U}$ that can represent any other rewrite theory $\mathcal{R}$ as a term $\overline{\mathcal{R}}$ (including when $\mathcal{R} = \mathcal{U}$) so that if $\mathcal{R}$ proves some sentence α, then $\mathcal{U}$ proves that $\mathcal{R}$ proves α.

The sentences α of rewriting logic are rewriting relations between terms t, and t', written as $t \rightarrow t'$, in which t' is reached by t by zero or more steps, according to the deduction rules of rewriting logic's calculus [12]. Formally, we express the universality of a theory $\mathcal{U}$ as:

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

where $\overline{t}$ and $\overline{t'}$ are the meta-representations of t and t', respectively.

As we said before, $\mathcal{U}$ is itself representable as a term, giving rise to the so-called *reflective tower*, with arbitrary levels of reflection.

In Maude, the universal theory $\mathcal{U}$ is implemented in the functional module `META-LEVEL`, which includes modules `META-MODULE` and `META-TERM`. The key functionality in these modules lies in the *descent* functions `metaParse`, `metaReduce`, `metaRewrite`, and `metaPrettyPrint`. We give next a summary of their functionality:

- `metaReduce(`M`,` t`)` is the metarepresentation of Maude's `reduce` command, which reduces a term to its canonical form. It attempts to reduce the term t using the operations and equations defined in the module M.
- `metaRewrite(`M`,` t`,` b`)` is the metarepresentation of `rewrite` command in Maude. It rewrites the term t according the rules in module M, after reducing t, until no more rewrite rules are applicable, or a user-defined number of rewrites has been reached (via the b parameter).
- `metaParse(`M`,` QL`,` T`)` constructs a term of sort/kind T from an arbitrary input string, encoded as a sequence QL of quoted identifiers representing the input tokens. The function also has a parameter a functional module M that defines the signature of the terms to be parsed.
- `metaPrettyPrint(`M`,` t`)` is the inverse of `metaParse`, in which it converts a term t to a sequence of quoted identifiers that represent the object syntax of t, according to a signature defined in module M.

All this combined with the fact that rewriting logic is a general framework for the representation of logics, computer systems, and programming languages due to its general calculus, adds up to the possibility for the creation of *formal tools* [7] for these

concepts. This is implemented by way of a mapping from the concept being formalized into a rewriting logic theory, that is, a functional or system module in Maude. The reflective aspect provides us with the necessary support for mechanization of this process.

Formally, this is achieved by defining a translation $\Psi : \mathcal{L} \rightarrow \mathcal{R}$, which is implemented using the reflective capabilities as a translation between terms that represent programs in $\mathcal{L}$ to terms that represent rewrite theories in $\mathcal{R}$. Let us call $\texttt{Module}_{\mathcal{L}}$ a module that represents programs in language $\mathcal{L}$. What we want is, in fact, a function $\tau : \texttt{Module}_{\mathcal{L}} \rightarrow \texttt{Module}_{\mathcal{R}}$. Section 3 outlined how such a function might be defined in the case where $\mathcal{L} = \text{MSOS}$.

We now proceed to show how we added support for MSOS-SL modules in Maude. For that to happen, users must be able to input MSOS-SL programs to the Maude interpreter, execute MSOS-SL specifications, and list previously entered modules.

This was achieved by extending Full Maude, which is an application that makes heavy use of the reflective power of rewriting logic and Maude. Full Maude defines a rich module algebra which includes module hierarchies, parameterization, views, theories, module expressions, and object-oriented modules.

User input in Full Maude is handled by Maude's `LOOP-MODE` facility, a general input/output method that receives user input within parenthesis and converts it into a sequence of "tokens", represented as quoted-identifiers. `LOOP-MODE` also converts a given sequence of tokens to output. This sequence of tokens defines the formatting of the output, including spacing, coloring, etc. Using this input/output schema, user input enclosed in parenthesis is parsed into terms by the `metaParse` function. Informally speaking, as modules are processed via user input, they are inserted into a *database* of modules. Again, this is possible since Maude is a reflexive language and its modules may be metarepresented as *terms* in this database. Full Maude's module hierarchy has `Unit` as its topmost sort, with several subsorts of each type of module supported, such as `StrFModule` for functional modules, and `StrSModule` for system modules. For the execution of these modules, $\texttt{Module}_{\mathcal{L}}$ modules are compiled into system modules, as outlined before. The function `metaRewrite` is then used to rewrite arbitrary terms using the compiled module.

Full Maude defines the signature of the user input, such as module declarations, commands, in the module `META-FULL-MAUDE-SIGN`. The first extension point is then to create an extended version of that module, called `EXT-META-FULL-MAUDE-SIGN`, that adds to Full Maude the signatures of user input that the Maude MSOS Tool handles. This currently resumes to the syntax definition of MSOS-SL modules but may be extended to add commands that are specific to the MSOS domain.

Once the user input is parsed, it must be handled by a specific function defined in Full Maude. This mapping of user input and their corresponding handlers is created in the `DATABASE-HANDLING` module by way of an `EXT-DATABASE-HANDLING` module that has necessary mappings to deal with MSOS user input.

Parsed MSOS-SL modules are of the sort `StrMModule`, also a subsort of `Unit`, which are then compiled into system modules of the sort `StrSModule` that will be used for the execution and verification of MSOS-SL specifications. The compilation function is defined with the following signature, and implements the mapping defined in Section 3.

`convertMSOS` : `StrMModule` → `StrSModule`

Now, we may execute and verify a MSOS-SL specification by way of this compiled module. Also, we define a pretty printer for MSOS-SL specifications by extending Full Maude's `eMetaPrettyPrint` to support `StrMModule` modules.

6 Executing MSOS-SL

In this section we exemplify the use of the Maude MSOS Tool. First we execute the example on Section 4. Next we execute a concurrent program written in the Concurrent ML language (CML) [25], an extension of Milner's Standard ML [19] with concurrency features, exploring Maude's logic programming features.

6.1 Executing a let expression

For the execution of a simple **let** expression, we need to specify the contents of the label associated with that expression. For this simple example, only two components are needed: `Env`, the environment of bindings from identifiers to values, and `Val`, a component needed for the evaluation of pattern matching.

We shall not give the semantics of declarations as pattern matching among identifiers and values nor its specification in MSOS-SL. We refer to [21] for the latter and `http://www.ic.uff.br/~cbraga/losd/maude-msos-tool/` for the former.

In order to execute the **let** expression semantics in the Maude MSOS Tool, one may first provide a module in Maude that specifies the initial state for the semantic components, that is, the initial values for the label fields in the configuration. In our example we named the module `INTERPRETER` and it first imports the module `SML-SEMANTICS` with the language semantics and then defines the constant `init` of sort `Label` which declares a label with `env` and `val` indices initialized to as the empty environment (`< mt-env >`) and empty val (`< mt-val >`), respectively.

```
(mod INTERPRETER is protecting SML-SEMANTICS .
 op init : -> Label .
 eq init = { (env : < mt-env >), (val : < mt-val >) } .
 ops x y : -> Ide .
endm)
```

We now may execute the expression using Maude's `rewrite` command as follows:

```
Maude> (rew < let val x = $(10) in
                  let val y = x in y end
              end, init > .)

rewrite in TEST :
  < let ... end, init >
result Conf :
  < $(10), {(env : < mt-env >), (val : < mt-val >)} >
```

Numbers are represented here with syntax `$n$`, with `n` of the sort `Nat`, to avoid pre-regularity conflicts. This is a technicality and we refer to [6] for a detailed explanation.

6.2 Executing a concurrent program

We have specified in MSOS-SL a significant subset of CML, based on the specification given by Mosses in [21]. Due to space limitations we give here the formal definition of the concurrent primitives of CML. The complete specification can be found at `http://www.ic.uff.br/~cbraga/losd/maude-msos-tool/`. For each primitive we give its informal semantics and then its MSOS-SL specification.

We begin with the representation of processes. Processes in CML have the general form `proc(`PI`,`E`)`, where PI is a *process identifier* (pid), and E is a CML expression. As processes are created, they are joined in a *pool of processes*. A complete CML program consists of at least one process and is written as: `cml` P, where P is the pool of processes. Four primitive operations are defined: `spawn`, for the creation of new processes, `send` and `recv`, for sending and receiving information through a channel, and `channel`, that allocates new communication channels. Message-passing in CML is synchronous, that is, a process transmitting or receiving information blocks until the operation is completed. A successful communication (or synchronization) is defined when there is another process sending a value on a channel c while there is a process waiting to receive something on the same channel. In this case, we say that both processes *agree* on their synchronization operations.

Concurrency constructs have the following signature in MSOS-SL:

```
(msos CML-CONCURRENCY is
 including SML-IMPERATIVES .
 including PIDE .

 sorts Prog Procs .

 op cml_ : Procs -> Prog [ctor] .
 op _||_ : Procs Procs -> Procs [ctor comm assoc] .
 op proc : PIde Exp -> Procs [ctor] .

 ops spawn channel send recv : -> Value [ctor] .
sosm)
```

The concurrency constructs have been defined as constants of sort `Value`, the sort of final computed values such as naturals and tuples, because both CML and SML have the concept of *application of expressions* of the form $E_1\ E_2$. Therefore there are rules for the application of each concurrency primitive to the appropriate value.

The `_||_` operator is the syntax for the *pool of processes*, defined to be both commutative and associative. This gives this operator the semantics of a *mathematical multiset*, where elements have no fixed order.

The module `CML-CONCURRENCY` is `including` the module `SML-IMPERATIVES`, which specifies the semantics of a subset of Standard ML. It also includes the module that declares the sort `PIde`.

We now proceed to the specification of concurrency constructs in CML. We make use of two components: `PIdes`, a read-write component that keeps track of the pids that have been created, and `Create`, a write-only component with identity `nilc` and

binary operator `appendc`, that signals the creation of a new process by the `spawn` command.

```
read-write pides : PIdes .
write-only create : Create (nilc, appendc) .
```

Rule 14 specifies the semantics of the `spawn` construct. When `spawn` is applied to an anonymous function, it proceeds to: (i) allocate a new pid; (ii) register this new pid on the relevant component; (iii) signals the creation of a new process by moving the process formed by the new pid and the application of the anonymous function to the empty tuple to the `Create` component.

```
ctr (spawn f) = {(create' : C),
                 (pides : PDS), (pides' : PDS'), IIS} => PI
if PI := newPIde (PDS) /\ PDS' := addPIde (PDS, PI) /\
   C := new-create (proc (PI, (f tuple()))) .                (14)
```

The variable `IIS`, of sort `IIndexSet`, indicates that any unspecified components will remain unchanged by this transition. Also, `tuple()` is the final value obtained from the evaluation of an empty tuple, written as `()` in CML.

Rule 15 puts the recently spawned process in the pool, and behaves as follows: when the evaluation of an expression inside some process gives rise to a process p in the `Create` component, process p is moved from the `Create` component to the pool of processes. The `Create` component is then cleared. `Create` is actually declared as a list but it will never have more than one element at a time. The function `get-one` returns the contents of the component *only when* it contains one element.

```
ctr proc (PI1, E1) ={(create' : nilc), IS}=>
    proc (PI1, E'1) || P
if E1 ={(create' : C), IS}=> E'1 /\ P := get-one (C) .       (15)
```

If no process creation is signaled in the `Create` component, a process must continue to evaluate normally. This is specified by Rule 16.

```
ctr proc (PI1, E) ={(create' : nilc), IS}=> proc (PI1, E')
if E ={(create' : nilc), IS}=> E' .                          (16)
```

Rule 17 selects one process from the pool to step, that is, to evaluate. In Mosses' MSOS specification of CML there are two rules for the nondeterministic evaluation between `P1` and `P2`. In our MSOS-SL specification, since we have defined the `_||_` operator to be *associative and commutative*, only one rule is needed.

```
ctr P1 || P2 = X => P'1 || P2
if P1 = X => P'1 .                                           (17)
```

Process synchronization constructs need two components: `Channels`, a read-write component that keeps track of the created channels, and `Offers`, a write-only component that manages the synchronization of processes. These indices are declared in MSOS-SL as follows.

```
read-write chans : Channels .
write-only offer : Offers (nilo, appendo) .
```

Rule 18 for the allocation of channels is straightforward: it creates a new channel and inserts it in the `Channels` component.

```
ctr channel tuple() ={(chans : chs),
                      (chans' : chs'), IIS}=> ch
if ch := newChannel (chs) /\ chs' := addChannel (chs, ch) .  (18)
```

When a process executes a `send` or `recv`, it creates an *offer* that is added to the `Offers` component, signalizing its will to synchronize. Two types of offers are defined:

```
op snd : Channel Value -> Offer [ctor] .
op rcv : Channel -> Offer [ctor] .
```

Our MSOS-SL specification slightly differs from Mosses' MSOS specification in this point. In Mosses' specification, the `rcv` offer also has a `Value` parameter parameterized by a free variable that eventually will be *unified* with a value from the `snd` offer from another process. However, the Maude MSOS Tool is implemented on top of Maude version 2.1.1, the version available at this time of writing, and it does not have unification yet. Therefore our MSOS-SL specification uses a *placeholder* `recv-ph` that is to be rewritten at the right time by way of a *metafunction*. (We refer to [5] for the operational details on this approach.). The placeholder constructor is defined as follows.

```
op recv-ph : Channel -> Value [ctor] .
```

Rule 19 specifies that when a process wants to send a value `v` through channel `ch`, it adds an *offer* `snd(ch,v)` to the `Offers` component and rewrites to the empty tuple.

```
ctr send tuple(ch, v) ={(offer' : O), IIS}=> tuple()
if O := new-offer (snd (ch, v)) .                          (19)
```

When a process is ready to receive a value through channel `ch`, it adds an *offer* `rcv(ch)` to the `Offers` component and rewrites `recv ch` to the placeholder `recv-ph(ch)`. This is specified by Rule 20.

```
ctr recv ch ={(offer' : O), IIS}=> recv-ph (ch)
if O := new-offer (rcv (ch)) .                             (20)
```

For the synchronization effectively to occur, two offers must *agree*. For example: a $snd(c, v)$ agrees with a $rcv(c)$, but not with a $rcv(c')$ if $c' \neq c$. If two offers agree, then the function `agree-value` declared below returns the value v that parameterizes the send offer involved in the synchronization.

```
op agree : Offer Offer -> Bool .
op agree-value : Offer Offer -> Value .
```

Two processes may step at the same time if on that step both of them signalize offers that *agree* on their `Offers` component. When this is the case, the `update-recv` function updates the process receiving the value by replacing the *placeholder* for the sent value. The `Offers` of the remaining computation is set to be empty. Rule 21 specifies this case.

```
ctr P1 || P2 ={(offer' : nilo), IIS}=>
    P'1 || update-recv(P'2,v)
if P1 ={(offer' : O1), IIS}=> P'1 /\
   P2 ={(offer' : O2), IIS}=> P'2 /\
   o1 := get-offer (O1) /\ o2 := get-offer (O2) /\
   agree (o1, o2) /\ v := agree-value (o1, o2) .          (21)
```

Finally, Rule 22 only steps the entire pool of processes when the `Offers` component is empty, meaning that processes are not allowed to continue their execution if they are synchronizing.

```
ctr cml P ={(offer' : nilo), IS}=> cml P'
if P ={(offer' : nilo), IS}=> P' .                        (22)
```

As an example of application of these rules, consider the case where two processes, P_1 and P_2 try to send different values to another process, P_3. There are two possible outcomes: (i) P_1 successfully synchronizes with P_3 and the transaction completes. In this case, process P_2 will remain blocked forever trying to synchronize with P_3; and (ii) P_1 is blocked while P_2 manages to transmit the value. The following shows that example using the Maude `search` command on the rewrite theory produced by the Maude MSOS Tool using the CML MSOS-SL specification as input. Solutions 1 and 2 represents cases (i) and (ii), respectively.

```
(search exec (let val c = channel t()
              in  (spawn (fn x => send t(c, $(10))) ;
                   spawn (fn x => send t(c, $(11))) ;
                   recv c)
              end) =>! C:Conf .)

search in CML-TEST : exec(let ... end) =>! C:Conf .
Solution 1
C:Conf <- < cml(proc(pide(0),$(10)) ||
                proc(pide(1),empty-tuple) ||
                proc(pide(2),
                 let <[c,chn(1)]> in
                  let <[x,empty-tuple]> in
                   send tuple(chn(1),$(11))
                  end
                 end)), {...} >
Solution 2
C:Conf <- < cml(proc(pide(0),$(11))||
                proc(pide(1),
                 let <[c,chn(1)]> in
                  let <[x, empty-tuple]> in
                   send tuple(chn(1),$(10))
                  end
                 end) ||
                proc(pide(2), empty-tuple)), {...} >
No more solutions.
Bye.
```

7 Final remarks

This paper presented Maude MSOS Tool and the MSOS-SL language for the specification of formal semantics of programming languages in the Modular Structural Operational Semantics framework formally specified by means of a semantics preserving mapping from MSOS and rewriting logic [16].

The Maude MSOS Tool is a mature prototype that evolved from previous development of one of the authors together with Hæusler, Meseguer, and Mosses [1, 2]. Even though much have been done since the first implementation [1], performance is still an issue since the use of rules in the generated Maude theories produces a large number of states that slows down verification times. A translation to unconditional rewrite rules [26], using a continuation-passing style, that would significantly improve performance according to a prototype that we have developed for CML using this technique. (`http://www.ic.uff.br/~cbraga/losd/specs/cml-cps/cml.maude`) Another approach is the (automatic) use of equational abstractions [17], by means of annotations in the MSOS-SL specification that would allow the Maude MSOS Tool to produce equations instead of rules, therefore shortening down the state space.

The current mapping from MSOS to rewriting logic handles concurrency using interleaving. It is part of our future work to explore the true concurrency model available in rewriting logic [13] which is also related to the representation with unconditional rewrite rules.

Finally, continuing on the work started in [1], we are developing a new version of the Maude Action Tool which is a prototype system for the execution and verification of Action Semantics [20] specifications using the Maude MSOS Tool, following its MSOS specification given in [11].

After we begun the development of the Maude MSOS Tool, Peter Mosses has defined its own specification language for MSOS, MSDF, along with an interpreter, written in Prolog. (`http://www.brics.dk/~pdm/MSOS/`) We are working together with Mosses on a comparison of our two systems in terms of usability and efficiency and also to unify the specification languages.

8 Acknowledgments

Braga would like to thank Peter Mosses, José Meseguer, and Edward Hermann Hæusler for their collaboration on the development of the first attempt to map MSOS to rewriting logic, and to José Meseguer on the development of the new mapping from MSOS to rewriting logic, used on the Maude MSOS Tool.

The authors would like to acknowledge partial support from CNPq under process 552192/2002-3. Braga is also partially supported by CNPq under process 300294/2003-4 and Chalub would like to thank FGV/EPGE for its partial support.

References

1. Christiano Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, September 2001. `http://www.ic.uff.br/~cbraga`.

2. Christiano Braga, Edward Hermann Haeusler, José Meseguer, and Peter D. Mosses. Mapping modular sos to rewriting logic. In Michael Leuschel, editor, *12th International Workshop, LOPSTR 2002, Madrid, Spain*, volume 2664 of *Lecture Notes in Computer Science*, pages 262–277. Springer, September 2002.
3. Christiano Braga and José Meseguer. Modular rewriting semantics in practice. In Narciso Martí-Oliet, editor, *Proceedings of 5th International Workshop on Rewriting Logic and its Applications, WRLA 2004*, Eletronic Notes in Theoretical Computer Science. Elsevier, 2004. To appear.
4. Roberto Bruni and José Meseguer. Generalized rewrite theories. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Proceedings of 30th International Colloquium on Automata, Languages and Programming, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2003.
5. Fabricio Chalub and Christiano Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004. `http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics`.
6. Manuel Clavel, Francisco Durán, Steven Eker, Narciso Martí-Oliet, Patrick Lincoln, José Meseguer, and Carolyn Talcott. *Maude 2*. SRI International and University of Illinois at Urbana-Champaign, `http://maude.cs.uiuc.edu`, 2003.
7. Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999 Proceedings, Volume II*, volume 1709 of *LNCS*, pages 1684–1703. Springer-Verlag, 1999.
8. Francisco Durán and José Meseguer. An extensible module algebra for Maude. In *Electronic Notes in Theoretical Computer Science*, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.
9. Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In Fabio Gadducci and Ugo Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
10. J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.
11. Søren B. Lassen, Peter D. Mosses, and David A. Watt. An introduction to AN-2, the proposed new version of action notation. In *AS 2000, Proc. 3rd International Workshop on Action Semantics, Recife, Brazil*, number NS-00-6 in Notes Series, pages 19–36. BRICS, Dept. of Computer Science, Univ. of Aarhus, 2000. Available at http://www.brics.dk/ pdm/papers/LassenMossesWatt-AS-2000/.
12. Narciso Martí-Oliet and José Meseguer. *Handbook of Philosophical Logic*, volume 61, chapter Rewriting Logic as a Logical and Semantic Framework. Kluwer Academic Publishers, second edition, 2001. `http://maude.cs.uiuc.edu/papers`.
13. José Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.
14. José Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *WADT'97*, volume 1376, pages 18–61. Springer, 1998.
15. José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. Manuscript. `http://maude.cs.uiuc.edu/papers`.
16. José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In *AMAST'04, Proc. 10th Intl. Conf. on Algebraic Methodology and Software Technology, Sterling, UK*, volume 3116 of *LNCS*, pages 364–378. Springer, 2004.

17. José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational abstractions. Submitted for publication, `http://maude.cs.uiuc.edu/papers`, 2003.
18. Robert Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer Verlag, 1980.
19. Robert Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (Revised)*. MIT Press, 1997.
20. Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
21. Peter D. Mosses. Fundamental concepts and formal semantics of programming languages – an introductory course. Technical report, University of Aarhus, Denmark, 2002.
22. Peter D. Mosses. Definitive semantics. Technical report, Warsaw University, 2003. `http://www.mimuw.edu.pl/~mosses/DS-03`.
23. Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.
24. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN - 19, Computer Science Department, Aarhus University, 1981.
25. John Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, June 1992. Technical Report TR 92-1285.
26. Grigore Roşu. From conditional to unconditional rewriting. In *Proc. 17th Int. Workshop on Algebraic Development Techniques (WADT 2004)*, 2004.