Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 133–146
www.elsevier.com/locate/entcs

Maude MSOS Tool

Fabricio Chalub[1] Christiano Braga[2]

Instituto de Computação
Universidade Federal Fluminense
Niterói, Brazil

Abstract

Modular structural operational semantics (MSOS) is a new framework that allows structural operational semantics (SOS) specifications to be made modular in the sense of not imposing the redefinition of transition rules, which is the case in SOS specifications, when an extension is made. Maude MSOS tool (MMT) is an executable environment for MSOS implemented in Full Maude as a realization of a semantics-preserving mapping between MSOS and rewriting logic (RWL). The modular SOS definition formalism (MSDF) is the specification language supported by MMT. MSDF syntax is quite close to MSOS mathematical notation and user-friendly by allowing several syntactic components to be left implicit. MMT joins the support for modularity with a user-friendly syntax together with the efficient execution and analysis of the Maude engine. We have used MMT in several different examples from programming languages semantics and concurrent systems. This paper reports on the development of MMT and its application to these two classes of specifications.

Keywords: Modular SOS, Modular SOS Definition Formalism, Rewriting Logic, Maude

1 Introduction

Structural operational semantics (SOS) [16] is a well known formalism for the specification of programming language semantics (e.g. [16]) and concurrent systems (e.g. [10]). However, an important aspect of specifications [12] was left open by Plotkin in his seminal lecture at Aarhus: modularity.

Plotkin's framework requires that transition rules in a given specification, handling certain semantic information, to be *modified*, when extensions to the specification alter the semantic information handled by the transition rules. That is, SOS is *not* modular. For instance, the transition rules of a specification of functional constructs for a programming language, that has an environment as a semantic component, have to be modified when the specification is extended with imperative

[1] Email: fchalub@ic.uff.br
[2] Email: cbraga@ic.uff.br

doi:10.1016/j.entcs.2007.06.012

constructs, that has a store as semantic component. This modification is necessary since the extended configuration has another component for the mapping from memory locations to values, that represent the store.

Plotkin himself declares in [16]: "As regards to modularity, if we get the other things in a reasonable state, then current ideas for imposing modularity on specifications will prove useful". Quoting Mosses [12], as opposed to support to modularity, "the other things" seem to be in good shape.

Mosses's modular structural operational semantics (MSOS) [12] solves the modularity problem in SOS specifications. The intuition is elegantly simple: given a label in a transition, a semantic component becomes a value bound to an index within the label which has a record-like structure, *encapsulating* the semantic information. Moreover, the record structure is extensible, therefore transition rules may refer to the label indices relevant to the language construct they specify. Transition rules in MSOS are then specified *once and for all*. For instance, transition rules for functional constructs only mention the environment and transition rules for imperative constructs relate the language constructs to a store. Therefore, in the extension with imperative constructs, the transition rules for the functional constructs are *not modified*.

The aim of this paper is to report on an effort for building and applying an executable environment for MSOS specifications: the Maude MSOS Tool (MMT).[3] MMT is a formal tool [4] implemented in Full Maude [5] as a realization of a semantics-preserving mapping between MSOS and rewriting logic (RWL) [8]. MMT supports the modular SOS definition formalism (MSDF) [11,3], a concrete syntax for MSOS, developed by the first author and Mosses, which appears to be quite suitable for such modular specifications, representing closely the mathematical notation of MSOS in [12]. We have used MMT to specify and analyze several programming language semantics specifications and concurrent systems. The first example is a set of MSDF specifications for *basic semantic primitives*, created by Mosses, known as Constructive MSOS (see e.g. [13]). The semantics of a programming language is given by a transformation from the programming language's syntax to Constructive MSOS primitives. We used Constructive MSOS to give executable (and analyzable) semantics for two programming languages in two different paradigms, namely functional and object-oriented. The second example is the semantics of a "lazy evaluation" functional programming language named Mini-Freja [15]. The third and fourth examples come from concurrent systems, and are, respectively, CCS and several distributed algorithms: Lamport's bakery algorithm, dining philosophers, leader election on an asynchronous ring, Peterson's solution for mutual exclusion, mutual exclusion with semaphores, and the thread game. The implementation of MMT and its many applications,[4] constitute the contributions reported by this paper.

The remainder of this paper is organized as follows. Section 2 discusses MSOS main features. Section 3 recalls the formal foundations of MMT. Section 4 discusses

[3] http://maude-msos-tool.sourceforge.net/

[4] MMT specifications can be downloaded from http://maude-msos-tool.sf.net/specs.

the implementation of MMT. Section 5 outlines the case studies we have developed using MMT. We conclude the paper in Section 6 discussing related efforts within the RWL community and the SOS community and describing our next steps.

2 Modular SOS

This Section presents key MSOS concepts and adapts material from [12].

The key to MSOS modularity lies on its transition labels which *encapsulate* the semantic information. MSOS labels are structured as *records* with each index in a record typed as *read-only*, *read-write* or *write-only*. A read-only index holds a value that does not change in a transition. Read-write indices are always declared in *pairs*: an index i represents information before the transition and index i' represents information after the transition. A write-only index, always declared "quoted" as in a side-effected read-write index, holds a value that may only be updated in a transition. (Thus, quoted indices always refer to information *after* a transition.) If we consider the specification of programming languages semantics, a read-only index holds an environment-like structure, a read-write indicex holds a memory store-like structure, and a write-only index holds (a free monoid of) emitted information, such as output or synchronization signals.

Let us illustrate this. The following SOS transition rule specifies (part of) the semantics for a **let** expression, typical in functional languages,

$$\frac{\rho_1[\rho_0] \vdash e \longrightarrow e'}{\rho_1 \vdash \textbf{let } \rho_0 \textbf{ in } e \longrightarrow \textbf{let } \rho_0 \textbf{ in } e'}$$

where ρ_0 and ρ_1 are sets of environment *bindings*, the SOS expression $\rho_1[\rho_0]$ means that the set of bindings ρ_1 is overridden by ρ_0, and e and e' are expressions in the object language being specified. Let us extend this specification with imperative assignments to memory locations given by the following transition rule,

$$\rho \vdash \langle l := v, \sigma \rangle \longrightarrow \langle \textbf{skip}, \sigma[l \mapsto v] \rangle$$

where l is a memory location, v is a (storable) value, σ is the memory *before* the transition, $\sigma[l \mapsto v]$ is the memory *after* the transition which is simply σ updated with the pair $l \mapsto v$, and **skip** is the identity of command composition. It should be clear now that the assignment extension implies the modification of the original transition rule for the **let** expression, since an assignment requires memory locations.

The following MSOS transition rule specifies (part of) the semantics for a **let** expression,

$$\frac{e \multimap\{\rho = \rho_1[\rho_0], \ldots\}\rightarrowtail e'}{\textbf{let } \rho_0 \textbf{ in } e \multimap\{\rho = \rho_1, \ldots\}\rightarrowtail \textbf{let } \rho_0 \textbf{ in } e'} \quad (1)$$

where the metanotation '$\multimap\{i = v, \ldots\}\rightarrowtail$' represents a labeled transition with 'i' any given index, 'v' a (general) value bound to 'i', and the ellipsis '...' representing the (same) remainder of the label, whose values *may* change. The remaining variables are typed as before. The index ρ is a read-only index capturing the environment. The meaning of an assignment could be given by the following transition rule,

$$l := v \multimap\{\sigma = \sigma_0, \sigma' = \sigma_0[l \mapsto v], -\}\rightarrowtail \textbf{skip} \quad (2)$$

where σ is a read-write index and '$-$' means that the remainder of the label is unobservable, that is, it can not change in such a transition. MSOS uses a "prime notation" to represent the value of an index *after* a transition. Therefore, the memory *before* the transition is given by the unprimed σ and the memory *after* the transition is given by the *primed* σ. Note that with the MSOS formulation, the transition rule for **let** need not to be modified, since the new rule for assignments simply range over the new index σ, distinct from index ρ, the one used by the **let** transition rule.

A computation in MSOS is a sequence of adjacent transitions with composable labels. Given two transitions $\gamma_1 \multimap\alpha_1\rightarrow \gamma_1'$ and $\gamma_2 \multimap\alpha_2\rightarrow \gamma_2'$, with configurations γ_1, γ_1', γ_2, γ_2' and labels α_1 and α_2, they are adjacent iff $\gamma_1' = \gamma_2$. Label composability is defined by cases on each possible index type: (i) for a read-only index i, $(\alpha_1;\alpha_2).i = \alpha_1.i = \alpha_2.i$; (ii) for read-write pair of indices i and i', $(\alpha_1;\alpha_2).i = \alpha_1.i$ and $(\alpha_1;\alpha_2).i' = \alpha_2.i'$, with $\alpha_1.i' = \alpha_2.i$; (iii) for a write-only index i', $(\alpha_1;\alpha_2).i' = \alpha_1.i' \cdot \alpha_2.i'$, where '$\cdot$' is the binary operation of the free monoid bound to i'.

To conclude our brief presentation of MSOS, let us define generalized transition systems (GTS), the models of MSOS specifications. A GTS is a tuple $\langle \Gamma, \mathcal{A}, \longrightarrow, T\rangle$, such that $\mathcal{A}$ is a category (named label category) with morphisms A, and $\langle \Gamma, A, \longrightarrow, T\rangle$ is a labeled terminal transition system [5], with Γ the set of configurations, $\longrightarrow \subseteq \Gamma \times A \times \Gamma$ the transition relation, and $T \subseteq \Gamma$ the set of terminal states.

3 RWL as a Semantic Framework for MSOS

In this Section we briefly recall the formal foundations of Maude MSOS Tool (MMT) [8], that is, MMT as an implementation of a formally defined semantics-preserving mapping from MSOS to RWL. The focus is on the intuition behind the ideas and not on the actual formalizations, which can be found in the cited references. The following paragraphs outline the main aspects of the mapping to help a proper reading of Section 4, as follows: the mapping from MSOS to RWL, Maude as

[5] A labeled terminal transition system is a labeled transition system where the terminal configurations are distinguished.

a formal metatool, and MMT as one such formal tool. We use Maude as a concrete syntax to define RWL theories.

The mapping from MSOS to RWL was defined by one of the present authors together with Haeusler, Meseguer, and Mosses [1] and then further developed jointly with Meseguer. The mapping produces, for each MSOS specification, a rewrite theory that includes a *record* theory, defined in the 'RECORD' module, applying a technique named modular rewriting semantics (MRS) [2]. A record is essentially a set of index-value pairs, with non-duplicated indices, that captures the notion of MSOS labels, with each record field declared by a membership axiom, such as 'mb (rho : E) : Field .', with 'E' of sort 'Env', a subsort of 'Component', and 'rho' a constant of sort 'Index'.

```
fmod RECORD is
sorts Index Component Field PreRecord Record .
subsort Field < PreRecord .
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] -> [Bool] .
var I : Index . vars C C' : Component . var PR : PreRecord .
eq duplicated((I : C),(I : C'), PR) = true .
cmb {PR} : Record if duplicated(PR) =/= true .
endfm
```

A *transition rule* is represented as a *conditional rewrite rule* in the generated rewrite theory, one of the choices proposed in [7]. Also, the generated rewrite rules should mimic operational semantics transitions. Therefore, RWL's inference rules for reflexivity, transitivity and congruence should *not* apply. This is controlled by the 'step' rule defined in the 'RCONF' module.

```
mod RCONF is protecting RECORD .
sorts Program Conf .
op <_,_> : Program Record -> Conf [ctor] .
op {_,_} : [Program] [Record] -> [Conf] [ctor] .
op [_,_] : [Program] [Record] -> [Conf] [ctor] .
vars P P' : Program .  vars R R' : Record .
crl [step] : < P , R > => < P' , R' > if { P , R } => [ P' , R' ] .
endm
```

The 'step' rule requires that all the transitions from an MSDF specification, which includes the 'RCONF' module, are represented as rewrites of the form $\{t\} \to [t']$. Thus, an MSOS computation is represented as a sequence of rewrites of the form $\langle t_1 \rangle \to \langle t_2 \rangle \to \cdots \to \langle t_n \rangle$, where t, t', t_1, t_2, and t_n are terms constructed out of the MSDF configurations and labels, as described in the next paragraph. The reflexivity inference rule is not applicable because the premise of the 'step' rule forces a change in the term. The rewrite cannot be transitive because there is no transition rule with a square bracket constructor on the left-hand side. The congruence inference rule is not applicable because subterms of t are not in the form $\{t\}$, required by the rewrite rules. (Of course, we assume that the configuration constructors, declared in the signature of module 'RCONF', do not appear as subterms of t, otherwise a proper renaming of the configuration constructors would be required.)

Note that the labeled transition rules induce a (labeled transition) relation with elements structured as *triples*, with the label being its second projection, as opposed to rewrite rules that induce a (rewriting) relation with *pairs* as elements. To produce an element of the rewriting relation from an element from the labeled

transition relation, the label from the transition relation is split into its *pre* and *post* projections, which are defined below. An element of the rewriting relation, built from an element from the labeled transition relation, is then given by a pair of pairs: the first pair is given by the first projection of the element from the labeled transition relation (e.g. the program syntax), and the pre-projection of the label, (e.g. the pre-projection of the environment); and the second pair is given by the second projection of the element from the labeled transition relation (e.g. the program syntax), and the post-projection of the label (e.g. the post-projection of the environment).

The pre- and post-projections are defined interpreting the MSOS label category (see Section 2) as a *pre-order*, with ordering relations defined for each possible index *type* in a label, that is, read-only, read-write and write-only as follows:

- both the pre- and post-projections of a read-only index yield the value bound to that index in a label;
- the pre-projection of a read-write index is the first projection of the pair bound to that index in a label and the post-projection of the index is the second projection of the value bound to that index;
- we consider a *trace* semantics for write-only indices, thus given a write-only index in a labeled transition in:
  - a *conclusion* of a transition rule, the pre-projection of the index is the *prefix* of the monoid bound to the index, and the post-projection is given by appending the value *emitted* by the transition, in the given index, to the value produced by the pre-projection;
  - a *premise* of a transition rule, the pre-projection is the identity element of the monoid and the post-projection is calculated in the same way as for the labeled transition in the conclusion of a transition rule. The identity element is used because we want only the information emitted at this index, in this transition, and not the complete trace.

The same rules for the calculations of pre- and post-projections apply to a label resulting from the composition of two other labels.

The mapping from MSOS to RWL is *semantics-preserving* in the precise sense of a *bisimulation* [8] between the models of MSOS specifications (GTS from Section 2) and the rewrite theories generated by the mapping from MSOS to RWL, that is, a rewrite theory with the rewriting relation representing small-step transitions as explained above.

Maude has been shown to be a formal metatool [4], which means that an executable environment can be built in Maude for a given concept (e.g., logic, specification language or model of computation) once a proper mapping is defined between this concept and rewriting logic. Such a mapping is then implemented as a (transformation) *metafunction* in Maude that, given a term in the signature of the module M representing the given concept, produces a term in the signature of Maude modules. Moreover, Full Maude may be used to create such an environment. Full Maude endows the Maude system with an *extensible* module algebra. In order to create the

executable environment, Full Maude should be extended with the above mentioned module M and metafunction. (Due to space limitations we do not detail the process of extending Full Maude.)

MMT is a formal tool implemented using the mapping from MSOS to RWL recalled from [8] in this Section as the transformation metafunction, with *modular SOS definition formalism* (MSDF) as concrete syntax for MSOS and Maude for RWL. Section 4 continues this paper exemplifying MSDF syntax and how MSDF specifications are transformed into Full Maude system modules.

4 Maude MSOS Tool

4.1 MSDF

MSDF is a concrete syntax for MSOS developed by the first author and Mosses. This Section describes the main elements of MSDF (modules, module inclusions, syntax definitions, label definitions, and transitions) by means of an example specification for **let** expressions, where values are integers.

MSDF modules begin with the string '`msos`' and end with '`sosm`'. Module inclusion can be implicit or explicit. An MSDF module M implicitly includes other modules that declare sets used in the declaration of datatypes and labels of M. Explicit inclusion of other modules is done using the Prolog-like '`see`' syntax. In our example, we declare a module '`LET`' that implicitly includes the specification for set '`Int`', since our **let** expressions only handle integers as values.

Syntax declarations are made using *sets* and *functions* on sets. It is also possible to use parameterized sets, such as tuples, lists, and maps. To illustrate syntax declarations in Backus-Naur form (BNF), labels, and transitions, we use the **let** example from Section 2. For instance, the module '`LET`' defines the **let** construct using mixfix syntax, and a Backus-Naur Form (BNF) style, as a function from `Dec` × `Exp` to `Exp`, declared as follows:

```
Exp ::= let Dec in Exp | Int | Id .
```

For each BNF declaration, two derived sets are implicitly declared for each set s in the declaration: 's`*`' and 's`+`' for sequences and non-empty sequences of elements from s, respectively. (The '`LET`' module, however, does not use any of these derived sets.) Other parameterized sets are defined with the following syntax: '`(`s`)List`' defines finite lists, '`(`s`)Set`' for finite sets, and '`(`s`,`k`)Map`' for finite maps from s to k. The '`LET`' module declares a map from identifiers to integers to define the set '`Env`', representing the environment, declared as follows:

```
Env = (Id, Int)Map .
```

Regarding label declaration, the indices of the components define a field to be read-only, read-write, or write-only using a "prime notation", as explained in Section 2: if there is a single, unprimed index, then the field defines a read-only component; an index that appears both unprimed and primed defines a read-write component with both components holding the same type of values; and a single primed index defines a write-only component. The ellipsis syntax '`...`' means that

the label declaration may be further extended with new components. In module 'LET', the declaration for the label with an environment component is as follows:

```
Label = { env : Env, ... } .
```

The transitions in MSDF should operate on *typed* value-added syntactic trees. Therefore the *type* of the term to be matched against should also be specified using the operator ':'. The type of the right-hand side of a transition is assumed to be the same as the type of the left-hand side. This is useful, for example, in languages where separate environments are provided for values and closures (e.g., Common Lisp). By using typed syntactic trees, the rule for the lookup of identifiers might use its type (a value-bound identifier or a closure-bound identifier) to choose the correct environment to use.

The syntax for transition rules in MSDF closely represents mathematical notation for inference rules. The premise is a (comma separated) conjunction of labeled transitions, predicates or Maude-like matching equations. As an example of a MSDF transition, the following transition rule specifies Rule 1, on Section 2.

```
    Exp -{env = (Env1 / Env2), ...}-> Exp'
-- ------------------------------------------------------------
    (let Env1 in Exp) : Exp -{env = Env2, ...}-> let Env in Exp' .
```

In MSDF there is no explicit variable declaration: every set name defines the prefix of a variable that may be "primed" or may end with a number, such as 'Exp'' and 'Env1', respectively. The 'Exp' after the colon on the left-hand side of the conclusion indicates the exact type of the term that will be matched by this transition.

4.2 Compiling MSDF into Full Maude

As exemplified in Section 4.1, an MSDF module has four components: a module inclusion section, syntax and datatype definition section, label declaration section, and transition rules declaration section. A high-level view of the compilation into Full Maude system modules is outlined as follows: 'see' declarations are transformed into module inclusions and implicit module inclusions are solved; syntax and datatype declarations are transformed into an equational theory; the label declaration is transformed into an extension of the 'RECORD' theory, and transition rules are transformed into conditional rewrite rules, with the label expressions transformed into record expressions as explained in Section 3. The remainder of this Section outlines the transformation for each component, and explains how the MSDF module LET in Section 4.1 is transformed into a Maude system module. The complete transformation details can be found in [3].

Compilation of module inclusions. The module inclusions in MSDF are converted directly into Full Maude inclusions. Implicit module inclusions are solved by searching Full Maude's database for the module name that declares each referenced set in the syntax declarations and label declarations of an MSDF module. The search is implemented as a meta-function that inspects the signatures of the meta modules in Full-Maude's database.

The Maude system module generated from the LET MSDF specification imports several modules, including views for automatically generated sequences of the MSDF

sets, a view for the finite map representing the environment, and a module for built-in integers.

Compilation of syntax definitions and datatypes. The syntax declarations in the MSOS specification are used to generate the signature of the Full Maude module, by converting each set declaration into a sort declaration, each subset inclusion into a subsort inclusion, and each function declaration into an operator.

Parameterized sets in MSDF—lists, sets, and maps—are converted into *parameterized sorts* in Full Maude. For each parameterized type, there is a built-in parameterized Full Maude functional module that implements the functionality of the relevant datatype. For example, tuples are transformed into an instance of module '`SEQUENCE (X::TRIV)`', which defines parameterized sorts '`Seq(X)`' and '`NeSeq(X)`'.

Each function declaration 's `::=` $f\ s_1\ s_2\ \cdots\ s_s$' in the set of functional symbols in the syntax declaration of a MSDF specification, compiles into an operator '$f : s'_1 \times \cdots \times s'_n \to s'$', where each set s_i is converted into a sort s'_i according to the compilation rules for the implicit sets into parameterized sorts.

Any attribute in f, such as `assoc` or `comm`, is moved verbatim to the generated operator. The function f may be in mixfix format. In this case, the functional symbol is constructed by keeping all the tokens beginning with lowercase identifiers and substituting all tokens beginning with uppercase letters by underscores. The sorts in the domain of the generated operator are named after the tokens beginning with uppercase letters in f.

To avoid preregularity problems, we chose not to subsort '`Program`' (from the '`RECORD`' module in Section 3) with sorts generated from the syntax of an MSDF module. (The same for the '`Component`' sort. See the paragraph **Compilation of label declarations** below.) This implies a special treatment of the '`step`' rule, as follows. To generate the configuration constructors for the step rule, the transformation function creates a set S_{max} with the top sorts from each connected component, induced by the subsort relations, generated from the syntax declaration section of a MSDF specification. For each sort $s \in S_{max}$ the following operators are declared in the signature of the generated Full Maude module: '`<_,_>` : $s \times$ `Record` $\to$ `Conf`', for the '`step`' constructor, '`{_,_}` : $s \times$ `Record` $\rightharpoonup$ `Conf`', for the left-hand side configuration constructor and '`[_,_]` : $s \times$ `Record` $\rightharpoonup$ `Conf`' for the right-hand side configuration constructor, both declared as partial functions. (Recall from Section 3 that an MSOS computation is represented as a sequence of rewrites of the form $\langle t_1 \rangle \to \langle t_2 \rangle \to \cdots \to \langle t_n \rangle$, where t_1, t_2, and t_n are terms constructed out of the MSDF configurations and labels.) Subsort overloading allows the declaration of these operators only for the top sorts.

A typed syntactic tree used in transitions is represented by a pair 't `:::` q', where t is the term representing the syntactic tree and q is a quoted-identifier that represents the sort. This quoted-identifier is the name of the sort prefixed with a single quote, such as '`'Exp`'. For each sort $s \in S_{max}$ an operator '`_:::_` : $s \times$ `Qid` $\to s$' is declared in the signature of the generated Full Maude system module.

The step rules are then generated for each sort $s \in S_{max}$:

$$\textbf{crl}\ \texttt{<}X ::: qid(s), R\texttt{>} \to \texttt{<}X' ::: qid(s), R'\texttt{>}$$
$$\textbf{if}\ \{X ::: qid(s), R\} \to \texttt{[}X' ::: qid(s), R'\texttt{]} \quad [step]$$

where X, X' are variables of the sort s, and R, R' are variables of the sort '`Record`'. The function $qid(s)$ converts a sort name into a quoted-identifier.

The signature of Maude system module generated from the '`LET`' MSDF specification contains sort declarations for every non-terminal on the right-hand side of the BNF declaration, such as '`Dec`' and '`Exp`', subsort declarations representing the alternatives in the BNF declarations, such as the one between '`Exp`' and '`Int`'. The same is done for the automatically generated sorts, such as '`Seq(Env)`', sequence for '`Env`', and '`Seq(Dec)`', the sequence of '`Dec`'. MSDF function declarations are transformed into operations, such as '`_"="_ : Id Int -> Dec`'. Operators for MRS configurations (e.g. '`<_‘,_> : Dec Record -> Conf`'), typed syntax-trees (e.g. '`_:::_ : Dec Qid -> Dec`'), field operators (e.g. '`_=_ : [Index] [Env] -> [Field]`') and '`step`' constructors (e.g. ''`[_‘,_‘] : [Dec] [Record] -> [Conf]`', and ''`{_‘,_‘} : [Dec] [Record] -> [Conf]`') are also declared. An example of a '`step`' rule is the following one, for '`Dec`'.

```
crl < P@:Dec ::: 'Dec,R:Record > => < P'@:Dec ::: 'Dec,R':Record >
if {P@:Dec ::: 'Dec,R:Record}=>[P'@:Dec ::: 'Dec,R':Record] [label  step] .
```

Compilation of label declarations. MSDF label declarations are transformed into an equational theory that extends the '`RECORD`' theory. To avoid preregularity problems, '`Component`' is not subsorted and for each index-semantic component pair in a MSDF label declaration, a '`Field`' operator ('`_=_`') and '`duplicated`' equations are declared on '`RECORD`' configurations.

A word about how MSOS labels and records are represented algebraically is needed: MSOS labels are defined as purely abstract sorts '`Label`', which represents an entire label; '`ILabel`', which represents an entire, unobservable label; '`FieldSet`', which represents a subset of the fields of a label, and '`IFieldSet`', that represents an unobservable subset of the fields of a label. The equivalent MRS sorts are, respectively, '`Record`', '`PreRecord`', '`IRecord`', and '`IPreRecord`'.

Additional subsorts of '`Field`' and '`Index`' are also defined as they are necessary to represent the possible index types. For read-only fields, the sort '`ROField`' is used; for read-write fields, the sort '`RWField`' is used; and for write-only fields, the sort '`WOField`' is used. For the indices, the following sorts are defined: for read-only indices, the sort '`RO-Index`' is used; for read-write indices, both sorts '`Pre-RW-Index`' and '`Post-RW-Index`' are defined related to the unprimed and primed indices, respectively; and finally for write-only indices, the sort '`WO-Index`' is used.

The '`RECORD`' specification of Section 3 does not capture the label algebra described in the previous paragraph, as it is inteded to represent records (with semantic information) in general and not only MSOS labels. Labels in MSOS should be composable, as explained in Section 2. Therefore, while using our record theory to represent MSOS labels, one should extend it with equations that capture how records and indices compose. They are essentially the same equations from Section 2, adapted to the record structure. The pre-regularity argument (previously used for not subsorting neither sort '`Program`' nor '`Component`') also applies here, so instead of declaring three equations specifying how read-only, read-write and write-only indices compose in general, such equations are generated for each index in a label declaration.

For the `LET` specification the read-only index '`env`' is declared, together with a

membership equation allowing elements of sort 'Env' to be part of the record structure, also an equation specifying how to compose two read-only indices from two distinct labels is declared, and finally an equation that checks for duplicated indices is declared.

```
op env : -> RO-Index    [ctor] .
mb env = V@:Env : ROField .
eq (I:RO-Index = C:Env,PR1:PreRecord);(I:RO-Index = C:Env,PR2:PreRecord)
  = I:RO-Index = C:Env,PR1:PreRecord ; PR2:PreRecord .
eq duplicated(I:Index = C@:Env,I:Index = C'@:Env,PR:PreRecord)
  = true .
```

Compilation of transitions. Transition rules are compiled into conditional rewrite rules. Essentially, the compilation process handles a relation between three elements—the two syntactic trees and the label—being converted into a relation between MRS configurations, that is, tuples containing the syntactic tree and the MRS record.

Dealing with the syntactic trees is straightforward: the left-hand side of MSDF transitions become the first projection on the left-hand side configuration, the program part, and the same idea applies to the right-hand side. Recall that, in MSDF, the syntactic trees have an associated *type*; this typed syntactic tree is converted to tuples of syntactic trees and types constructed by the '_:::_' operator, as explained in Section 4.1. MSOS label expressions in transition rules are compiled following the pre and post projections as explained in Section 3.

The transition rule for evolving 'let' expressions, from Section 4.1, is transformed into the following conditional rewrite rule. Note that the ellipsis '...' are represented directly as variables, and that a fresh variable 'envVAR0:Env' is declared for the environment in right-hand side of the rewrite in the condition. Such variable declaration allows for expressions not in normal-form, such as 'Env1:Env / Env2:Env', to be used in the left-hand side.

```
crl {let Env1:Env in Exp:Exp ::: 'Exp,{env = Env2:Env,...:PreRecord}}
  => [let Env1:Env in Exp':Exp ::: 'Exp,{env =  Env2:Env,...':PreRecord}]
  if {Exp:Exp ::: 'Exp,{env =(Env1:Env / Env2:Env),...:PreRecord}}=>
     [Exp':Exp ::: 'Exp,{env = envVAR0:Env,...':PreRecord}] [label  none] .
```

Alternatively, a matching equations could have been used to capture the value of the expression 'Env1:Env / Env2:Env' in 'envVAR0:Env' and then place it as the value bound to index 'env' in both sides of the rewriting condition. This would enforce, locally in the rule, the fact that the environment should not change in a transition.

5 MMT in Practice

This Section outlines examples created to assess the capabilities of MMT. The complete descriptions can be found in [3]. The first example is the MSDF specification of Mosses's *Constructive MSOS* where the semantics of a programming language is expressed in terms of basic, abstract, constructs. The MSDF specification of Constructive MSOS was further used to define the semantics of a subset of Reppy's Concurrent ML (CML) and Appel's MiniJava. The second example is the MSDF specification of Mini-Freja, a normal-order language [15]. The third example is the specification and verification of CCS. The fourth example is the specification and

verification of many distributed algorithms.

Constructive MSOS is an abstract syntax for usual programming languages constructs. Our MSDF specifications for Constructive MSOS consist of 800 lines, divided into 74 modules that define abstract constructs and basic data types commonly found in programming languages. (For instance, there is a module `Cons/Abs/closure` with the BNF and transition rules for closure values.) The fine-grained modularization of the specification is only possible due to *encapsulation* of the semantic components provided by MSOS, a modularization that greatly improves *reusability*.

We wrote two specifications using Constructive MSOS to exemplify its reusability. The semantics of two languages, MiniJava and a subset of CML were developed based on a translation from their concrete syntax into Constructive MSOS abstract constructs. The semantics of a complete programming language, such as MiniJava, is given by a MSDF module that gathers all necessary constructs and defines an initial environment for the beginning set of bindings. The conversion from MiniJava to the abstract constructs is performed, in our implementation, by the SableCC parser generator. (An external parser was used for a fine-grained lexical control during the parsing process.)

The semantics for MiniFreja was created as an example of an MSDF specification of a *lazy evaluation* functional language, specified in big-step operational semantics. We did not use external parsers, or libraries, following [15] straight-forwardly, and implemented the complete pattern matching algorithm specified there. To test the specification we implemented and executed the sieve of Eratosthenes using "lazy lists," as in the original MiniFreja specification.

It is well known that SOS is a formalism not only used in the specification of programming languages, but also of concurrent systems [10]. The specification and analyses of CCS[6] was straight-forward, since it was originally specified in operational semantics. We also implemented and analyzed several distributed algorithms: Lamport's bakery algorithm, some variations of dining philosophers, leader elect ion on an asynchronous ring, Peterson's solution for mutual exclusion, mutual exclusion with semaphores, and a thread game, where two threads compete for a shared resource. Using MSDF and MMT, the algorithms can be specified independently from any particular *scheduling* strategy. Such a scheduling strategy may be modularly added later.[7]

These experiments gave us confidence that MSDF can express quite well MSOS, either in small-step or big-step styles. We managed to write and analyze straightforwardly non-trivial operational semantics specifications already written in MSOS mathematical style, operational semantics specifications not written in MSOS style, and specify, quite succinctly, several concurrent systems that were not originally specified in operational semantics. Moreover, we analyzed all specifications using Maude's built-in state search and model-checker within reasonable time frames, according to related literature. However, the tool needs improvement regarding

[6] The MSDF specification of CCS was developed together with Alberto Verdejo.

[7] http://www.ic.uff.br/~cbraga/losd/publications/modular-da.pdf

usability. At the moment, it requires some understanding of Maude and how MSDF specifications are represented in Maude in order to execute and analyze MSDF specifications and understand some of the errors reported by MMT.

6 Related and Future Work

The relationship between SOS and RWL was first established by Martí Oliet and Meseguer in [7], and further developed by Verdejo in his PhD thesis [17]. In [14] the metarepresentation of GSOS in rewriting logic is used to implement methods for checking the premises of some SOS meta-theorems (e.g., GSOS congruence) in the GSOS framework. By restricting themselves to structural operational semantics, these approaches lack support for modular specifications. Modularity can be achieved in RWL, and not only while representing SOS specifications, using MRS theories, as mentioned in Section 3. Rewriting logic semantics [9] (RLS), developed by Meseguer and Roşu, allows for modular specifications with true concurrency, following a stack machine model and continuations. Both MRS and RLS, however, have readability problems, which appear to be more serious in RLS. The specification of true concurrency and continuations remains an open problem in MSOS and, therefore, in MSDF.

Mosses's own MSOS Tool [11], implemented using Prolog, is the only alternative to MMT, at the moment, for writing MSOS specifications. The efficiency of the tool needs some improvement and, while it has support for tracing through execution paths via Prolog, it currently does not have the ability to model check specifications or any mechanism to combine it with other tools.

Other significant operational semantics tools include Hartel's LETOS [6] and Pettersson's RML [15]. LETOS (Lightweight Execution Tool for Operational Semantics) supports SOS (and denotational semantics) by generating Miranda code, has interesting pretty-printing facilities and support for non-determinism by means of lists. RML (Relational Meta-Language) supports natural semantics specifications generating efficient C code. Neither support any form of modular specifications, search, or model-checking.

Regarding future work, we plan to explore the combination of MMT with user-defined Maude tools, such as the strategy language interpreter (SLI) from Martí Oliet, Meseguer and Verdejo. A prototype for a combined tool with SLI, developed together with Verdejo, has been developed, allowing the execution and search on MSDF specifications to be guided by strategies.

Acknowledgement

The authors would like to thank Peter Mosses and Alberto Verdejo for their helpful comments on a draft of this paper, and the anonymous referees for their careful review. This work was partially sponsored by CNPq.

References

[1] C. Braga, E. Hermann Haeusler, J. Meseguer and P. D. Mosses. Mapping Modular SOS to Rewriting Logic. In Michael Leuschel, editor, *12th International Workshop on Logic Based Program Synthesis and Transformation, LOPSTR 2002*, volume 2664 of Lecture Notes in Computer Science, pages 262–277, Madrid, Spain, September 2003. Springer.

[2] C. Braga and J. Meseguer. Modular rewriting semantics in practice. *Electronic Notes in Theoretical Computer Science*, 117:393–416, 2005.

[3] F. Chalub. An Implementation of Modular Structural Operational Semantics in Maude. Master's thesis, Universidade Federal Fluminense, 2005. http://www.ic.uff.br/~frosario/dissertation.pdf.

[4] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In *World Congress on Formal Methods (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.

[5] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Escuela Tcnica Superior de Ingeniería Informática, 1999.

[6] P. H. Hartel. LETOS – A lightweight execution tool for operational semantics. *Software—Practice and Experience*, 29(15): 1379–1416, Sep 1999.

[7] N. Martí-Oliet and J. Meseguer. *Handbook of Philosophical Logic*, volume 9, chapter Rewriting Logic as a Logical and Semantic Framework, pages 1–87. Kluwer Academic Publishers, second edition, Nov 2002. http://maude.cs.uiuc.edu/papers.

[8] J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. In C. Rattray, S. Maharaj, and C. Shankland, editors, *In Algebraic Methodology and Software Technology: proceedings of the 10th International Conference, AMAST 2004*, volume 3116 of *LNCS*, pages 364–378, Stirling, Scotland, UK, July 2004. Springer.

[9] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Electronic Notes in Theoretical Computer Science*, 156 (1):27–56, 2005.

[10] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[11] P. D. Mosses. Fundamental Concepts and Formal Semantics of Programming Languages—an introductory course. Lecture notes, available at http://www.daimi.au.dk/jwig-cnn/dSem/, 2004.

[12] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.

[13] P. D. Mosses. A Constructive Approach to Language Definition. *Journal of Universal Computer Science*, 11(7): 1117-1134, 2005.

[14] M. Mousavi and M. A. Reniers. Prototyping SOS meta-theory in Maude. *Electronic Notes in Theoretical Computer Science*, 156 (1):135–150, 2005.

[15] M. Pettersson. *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer, 1999.

[16] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[17] J. A. Verdejo. *Maude como um marco semântico ejecutable*. PhD thesis, Universidad Complutense de Madrid, 2003.