# Maude MSOS Tool

Fabricio Chalub    Christiano Braga

July 11, 2005

## Contents

## 1 Introduction

This technical report describes the *Maude MSOS Tool* [3] (MMT), a programming environment for Modular SOS specifications. MMT is a formal tool in the sense of [5], implemented as a conservative extension of *Full Maude*, that compiles MSOS specifications into rewriting logic. The implementation is based on a mapping that was first described in [1] and later developed in [2]. The compilation into rewriting logic modules enables the use of Maude's execution and formal verification tools in MSOS specifications.

The syntax of modules accepted by MMT is based on the *Modular SOS Definition Formalism* (MSDF) described by Mosses in [11]. Both languages are similar enough to warrant the designation of the specification language of the MMT also as "MSDF." The minor differences that exist between the two are restricted to idiosyncrasies of the Maude parser. The "MSDF" designation henceforth refers to the version currently accepted by the MMT, except otherwise noted.

While MMT was designed with the primary objective of being a formal environment for the specification of programming languages, operational semantics, including MSOS, is an environment applicable on a wide variety of topics. For example, operational semantics was used in the development of several concurrency calculi, such as CCS [9] and the $\pi$-calculus [10]. However, given its main purpose, we opted for the exposition of the different MSDF constructions by motivating their use on the formal definition of some programming language `L`.

To give the definition of `L` is, in fact, to give the definition for each one of its $\mathtt{L}_\mathtt{c}$ constructions. It consists of the formal description of the *syntax* and *semantics* of $\mathtt{L}_\mathtt{c}$. For the syntax we follow the traditional approach and avoid the complications of concrete syntax by adopting an abstract version of $\mathtt{L}_\mathtt{c}$ (see, for example, [13, Section 1.4.1] and [11, Section 1.2.3] for a discussion on this). Let us call this abstract construction $\mathcal{L}_\mathtt{c}$. Also following usual practices, we describe the context-free grammar of $\mathcal{L}_\mathtt{c}$ using Backus Naur Form (BNF) notation.[1] The specification of a programming language abstract syntax is the subject of Section 3.2.

The semantics of the $\mathcal{L}_\mathtt{c}$ construction is defined by *transition rules*, with

[1]An interesting historical perspective on the name Backus Naur Form versus Backus Normal Form appears on Donald Knuth's collection of programming language papers [7].

*labels* containing the necessary semantic components. We discuss how labels and transitions are specified in MSDF in Sections 3.3 and 3.4, respectively.

Specifications may be organized into modules to allow the modular construction of a specification, something desirable not only for didactic purposes, but also as a sound engineering practice to cope with the complexity inherent on any large scale specification project. The use of modules is described on Section 3.1.

## 2 Notational conventions

On the following sections the grammar of MSDF constructions is specified using an extended BNF grammar, with terminals '`in teletype font and between single quotes`' and non-terminals specified in ⟨ sans-serif font and between angle brackets ⟩. Extended BNF notation is always used on an expression within meta-parentheses: '$(\mathrm{E})^*$' and '$(\mathrm{E})^+$' denote, respectively, zero or more, and one or more, repetitions of E; '$(\mathrm{E})^?$' denotes an optional expression.

Before we explain the grammar of MSDF's constructions we must discuss some lexical aspects. As an extension of *Full Maude*, we also make use of Maude's `LOOP-MODE`, and are restricted to its lexical analysis capabilities. In order to insulate this description from the technical details of Maude's own current lexical analysis, we use some predefined non-terminals in an abstract way: the first is ⟨ id ⟩, for *identifiers*, that follows Maude's definition of identifiers. As of Maude version 2.1.1 the lexical rules for identifiers are as follows (taken from [4]):

> Any finite string of ASCII characters such that:
>
> - it does not contain any white space;
> - the characters '`{`', '`}`', '`(`', '`)`', '`[`', '`]`', and '`,`' break a sequence of characters into several identifiers;
> - the back-quote character '`' is used as an escape character to indicate that a blank space or the special characters following it do not break the sequence.

Identifiers with the initial letter in uppercase are represented by ⟨ upper-id ⟩, while those with the opposite constraint—the initial letter in lowercase—are represented by ⟨ lower-id ⟩. With the current version of Maude it is not possible to specify this and other (e.g., that an identifier may consists only of letters) restrictions, usually stated as regular expressions, so they must be checked after the parsing is done.

Another predefined non-terminal is $\langle$ term $\rangle$ which represents *value added syntax trees* that appear in transition rules.

In Full Maude, all user input must be made between parentheses.

## 3 MSDF syntax

These Sections describe the elements of the MSDF specification language. Each Section begins with a formal exposition of the constructions, followed by an illustrative example, and ends with the details on usage.

### 3.1 Modules

$\langle$ module $\rangle \rightarrow$ '`msos`' $\langle$ id $\rangle$ '`is`' ($\langle$ see $\rangle$)$^*$ ($\langle$ declaration $\rangle$)$^*$ '`sosm`'
$\langle$ see $\rangle \rightarrow$ '`see`' $\langle$ name $\rangle$ ('`,`' $\langle$ name $\rangle$)$^*$ '`.`'

All modules in MSDF begin with the string '`msos`' and end with '`sosm`'. An MSDF module name is any identifier ($\langle$ id $\rangle$) accepted by Maude (Section 2). To include other modules, one uses the $\langle$ see $\rangle$ constructions, which is the keyword '`see`', followed by a list of modules identifiers, separated by commas; multiple $\langle$ see $\rangle$ lines are permitted for flexibility of the specification text, but they are only sugar for a single line of module importations.

The remainder of the module consists of a sequence of MSDF declarations, represented by the non-terminal $\langle$ declaration $\rangle$. In order to allow a flexible specification, the order does not matter, since all declarations are first collected before the compilation begins.

As an example of module inclusion, consider the following MSDF module, that defines a module '`PL-SYNTAX`' that, in turn, includes the modules that define the several different components of some fictitious language PL.

```
msos PL-SYNTAX is
 see PL-EXPRESSIONS, PL-DECLARATIONS .
 see PL-IMPERATIVES, PL-ABSTRACTIONS .
 see PL-CONCURRENCY .
sosm
```

As a general rule, in order to be used in a module an MSDF declaration must be previously defined in some other module (or in the module itself) and its declaring module must be explicitly included. However, it so happens that some inclusions may be better omitted not only for brevity, but also for clarity of MSDF specifications. It is not uncommon in programming language definitions for a basic set of modules to be needed in *most* of the other modules, since primitive constructions, such as commands and expressions are probably used by even the most advanced features; it would be tedious and error prone to force the user to explicitly declare these inclusions on each module that they are needed, specially when they are obviously needed.

The concept of "obvious need" is subjective and, to avoid confusion, the MMT has a simple rule for the omission of a module inclusion: all defining modules of sets that are used in a module are included by default. This includes not only sets used on the datatype definition part, but also the label declaration part.

For example, consider the following MSDF module that defines an abstract syntax of conditional expressions:

```
msos COND is
 Exp ::= if Exp then Exp else Exp .
sosm
```

The system automatically includes the module that contains the declaration of the set '`Exp`'. If this module is, say, '`EXP`', then the expanded module, without any implicit inclusions, would be:

```
msos COND is
 see EXP .

 Exp ::= if Exp then Exp else Exp .
sosm
```

It is important to emphasize that this rule is restricted to *sets* only; if a module defines a construction that uses other constructions, the defining modules must be explicitly included. Consider the following example, where a '`while`' is defined in terms of conditionals ('`if then else`'), and command sequencing ('`;`'). The modules that define these constructions ('`COND`', and '`SEQ`', respectively) had to be explicitly included, while the modules that define the sets '`Exp`' and '`Cmd`' were omitted.

```
msos WHILE is
  see COND, SEQ .

  Cmd ::= while Exp do Cmd .

  (while Exp do Cmd) : Cmd -->
   if Exp then (Cmd ; while Exp do Cmd) else skip .
sosm
```

## 3.2 Datatype definitions

⟨ setid ⟩ → ⟨ upper-id ⟩
⟨ opid ⟩ → ⟨ lower-id ⟩
⟨ dsetid ⟩ → ⟨ setid ⟩ | ⟨ setid ⟩'`*`' | ⟨ setid ⟩'`+`'
⟨ mixfix-fun ⟩ → ⟨ opid ⟩ | ⟨ dsetid ⟩
⟨ dec ⟩ → ⟨ setid ⟩ '`.`' | ⟨ setid ⟩ '`::=`' ⟨ dec-rhs ⟩ ('`|`' ⟨ dec-rhs ⟩)$^*$ '`.`'
⟨ dec-rhs ⟩ → ⟨ dsetid ⟩ | ⟨ mixfix-fun ⟩$^+$ ('`[`'⟨ attr ⟩'`]`')$^?$
⟨ param ⟩ → ⟨ dsetid ⟩ | '`(`'⟨ dsetid ⟩'`)List`' | '`(`'⟨ dsetid ⟩'`)Set`'
    | '`(`'⟨ dsetid ⟩'`,`'⟨ dsetid ⟩'`)Map`'
⟨ equiv ⟩ → ⟨ setid ⟩ '`=`' ⟨ param ⟩ '`.`'
⟨ declaration ⟩ → ⟨ dec ⟩ | ⟨ equiv ⟩

Although specified using BNF notation, the abstract syntax definition of a programming language in MSDF is in fact an algebraic datatype definition. This is also true in other operational semantics tools, including Mosses's own MSOS Tool, in Prolog and Hartel's LETOS, in Miranda. In this sense, non-terminals are sets, and sequences of non-terminals and terminals (⟨ mixfix-fun ⟩$^+$) are n-ary functions specified in the so-called *mixfix form* with the non-terminals representing the arguments and the terminals representing the function name in mixfix form. A function that contains only a single terminal is called a *constant*.

In MSDF, a ⟨ setid ⟩ is a *primitive set* name, an uppercase identifier (⟨ upper-id ⟩), such as '`Integer`'. This identifier must contain only letters, due to restrictions related to the formation of metavariables (see Section 3.4); a ⟨ dsetid ⟩ is a *derived set* name, since it represents a set automatically derived from the primitive sets. There are two possible derived sets: the set of possible-empty sequence of elements and the set of non-empty sequence of elements. For some set `s`, the former is written as `s*` while the later as `s+`.

The non-terminal ⟨ dec ⟩ declares either a new set, a subset inclusion declaration, or a function declaration, depending on the right-hand side (⟨ dec-rhs ⟩). If the right-hand side is a single ⟨ dsetid ⟩, then it is a subset inclusion (e.g., '`Exp ::= Value .`'). On the other hand, if the right-hand side is a sequence of lowercase and uppercase identifiers, it is interpreted as a mixfix function, as described above, e.g., '`Exp ::= if Exp then Exp else Exp .`'. A prefix form is, of course, a particular case of a mixfix function, e.g., '`Sys ::= parallel (Cmd, Cmd)`'. The optional attributes that may be specified to a mixfix function (the non-terminal ⟨ attr ⟩ enclosed between bracket parentheses) are discussed at the end of this Section.

Currently, MMT predefines two built-in sets: integers ('`Int`'), and booleans ('`Boolean`'), and their respective sequences '`Int*`', '`Int+`', '`Boolean*`', and '`Boolean+`'. The set '`Boolean`' contains two constants '`tt`' and '`ff`' that represents the values true and false, respectively.

Parameterized sets, defined by the non-terminal '`param`', are obtained using a modifier over one or two sets: given a set `s`, '`(s)List`' is the set of lists of elements of `s`; '`(s)Set`' is the set of finite sets of elements of `s`; given a second set `k`, '`(s,k)Map`' is the set of finite mappings from elements of set `s` to elements of `k`. Parameterized sets are not to be used directly, but by defining *equivalent sets* using the syntax of the non-terminal '`equiv`'.

The following fragment exemplifies the use of datatype definitions. Four sets are declared: '`Exp`', '`Value`', '`Cmd`', and '`Id`'. These sets represent, respectively, the expressions, values, commands, and identifiers of some programming language. Next, a subset inclusion is defined between '`Value`', and the built-in sets '`Int`' and '`Boolean`', that is, the set of values is augmented with elements from the sets of integers and booleans. Following, the abstract syntax of two constructions is defined in mixfix form: a conditional expression, and a looping command, which expects to receive a non-empty sequence of commands as its body. Finally, an equivalence between the set '`Env`' and the set '`(Id, Value)Map`'—a mapping between identifiers and values, traditionally used as an environment for bindings—is defined.

```
Exp .  Value .
Cmd .  Id .
Value ::= Int | Boolean .
Exp ::= if Exp then Exp else Exp .
```

```
Cmd ::= while Exp do Cmd+ .
Env = (Id, Value)Map .
```

Finally, mixfix functions in MSDF may have *attributes* to simplify notation. They are currently the attributes supported by the Maude system: '`associative`' (abbreviated '`assoc`'), used to define an associative binary function; '`commutative`' (abbreviated '`comm`'), used to define a commutative binary function; '`precedence n`' (abbreviated '`prec n`') that gives the precedence value of that function, where $n$ is an integer between zero and $2^{31}-1$, where a lower value indicates a tighter binding; and '`identity:t`' (abbreviated '`id:t`') that establishes the term `t` as the identity to the specified function. As in Maude, these attributes may be combined.

Attributes in MSDF datatype definitions permit the creation of a more flexible abstract syntax notation, while keeping its simplicity. The use of precedence values, for example, permits the specification of arithmetic functions with the right grouping on the abstract syntax, something that is usually done on the concrete syntax and then moved on to the abstract syntax explicitly. The Java Language Specification [6], for example, defines non-terminals '`UnaryExpression`', '`MultiplicativeExpression`', '`AdditiveExpression`', and '`RelationalExpression`' to cope with the several levels of precedence of these different expressions. Using MSDF, one may specify the same requirements using precedence and associative attributes as follows:

```
Exp .
Exp ::= Exp + Exp [assoc prec 10] .
Exp ::= Exp - Exp [assoc prec 30] .
Exp ::= Exp * Exp [assoc prec 20] .
Exp ::= Exp / Exp [assoc prec 20] .
Exp ::= Exp < Exp [assoc prec 10] .
Exp ::= Exp > Exp [assoc prec 10] .
Exp ::= Exp == Exp [assoc prec 10] .
```

By asking Maude to parse the expression '`e1 + e2 * e3 * e4 * e5 - e6`' using the declarations above, we obtain the following term, shown with parentheses to explicit the grouping order: '`(((e1 + e2) * (e3 * (e4 * e5))) - e6)`'.

## 3.3 Labels

$\langle$ label $\rangle \rightarrow$ ‘`Label={`’ $\langle$ field $\rangle$ (‘`,`’ $\langle$ field $\rangle$)$^*$ ‘`, ...} .`’
$\langle$ field $\rangle \rightarrow \langle$ index $\rangle$ ‘`:`’ $\langle$ derived $\rangle$
$\langle$ index $\rangle \rightarrow \langle$ lower-id $\rangle$
$\langle$ declaration $\rangle \rightarrow \langle$ label $\rangle$

Each MSDF module may contain at most one label declaration using the syntax of the non-terminal $\langle$ label $\rangle$. A label consists of a sequence of type declarations of $\langle$ field $\rangle$. Each field consits of an $\langle$ index $\rangle$ (which is a lowercase identifier, $\langle$ lower-id $\rangle$) and the type of its component ($\langle$ derived $\rangle$).

The indices of the components defines a field to be read-only, read-write, or write-only: if there is a single, unprimed index, then the field defines a read-only component, as in:

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

An index that appears both unprimed and primed defines a read-write component. Both components must be of the same type.

```
Label = { st : Store, st' : Store, ... }
```

Finally, a single, primed, index defines a write-only component. The only admissible type of write-only components are *sequences* of primitive sets, as in:

```
Label = { output' : Value*, ... }
```

If there are multiple label declarations in a single module, the last declaration is taken into account, while the others are ignored. Of course, a single label declaration may define several fields:

```
Label = { env : Env, st : Store, st' : Store,
          output' : Value*, ... }
```

## 3.4 Semantic transitions

⟨ transition ⟩ → ⟨ cond-transition ⟩ | ⟨ uncond-transition ⟩
⟨ cond-transition ⟩ → ⟨ cond ⟩ (‘`,`’ ⟨ cond ⟩)* (‘`[`’ ⟨ label ⟩ ‘`]`’)?

quad ‘`--`’ ⟨ step ⟩ ‘`.`’
⟨ uncond-transition ⟩ → (‘`[`’ ⟨ label ⟩ ‘`]`’)? ⟨ step ⟩ ‘`.`’
⟨ label ⟩ → ⟨ id ⟩
⟨ typed-term ⟩ → ⟨ term ⟩ ‘`:`’ ⟨ dsetid ⟩
⟨ step ⟩ → ⟨ typed-term ⟩ ⟨ relation ⟩ ⟨ term ⟩
⟨ cond-step ⟩ → ⟨ term ⟩ ⟨ relation ⟩ ⟨ term ⟩
⟨ relation ⟩ → ‘`-->`’ | ‘`==>`’ | ‘`-`’ ⟨ label-exp ⟩ ‘`->`’ | ‘`=`’ ⟨ label-exp ⟩
‘`=>`’
⟨ label-exp ⟩ → ‘`{`’ ⟨ field-exp ⟩ (‘`,`’ ⟨ field-exp ⟩)* ‘`}`’
⟨ field-exp ⟩ → ⟨ index ⟩ ‘`=`’ ⟨ term ⟩ | ⟨ composition ⟩ | ⟨ rest ⟩
⟨ rest ⟩ → ‘`...`’ | ‘`-`’ | ‘`X`’ | ‘`U`’
⟨ composition ⟩ → ⟨ rest ⟩ ‘`;`’ ⟨ rest ⟩
⟨ cond ⟩ → ⟨ eq ⟩ | ⟨ pred ⟩ | ⟨ inst ⟩ | ⟨ cond-step ⟩
⟨ eq ⟩ → ⟨ term ⟩ ‘`=`’ ⟨ term ⟩
⟨ pred ⟩ → ⟨ term ⟩
⟨ inst ⟩ → ⟨ term ⟩ ‘`:=`’ ⟨ term ⟩
⟨ declaration ⟩ → ⟨ transition ⟩

The ⟨ transition ⟩ non-terminal specifies how MSOS transitions are written in MSDF. Let us begin our description with unconditional transitions (⟨ uncond-trans ⟩), described by the non-terminal ⟨ step ⟩, with an optional label ⟨ label ⟩.[2] An unconditional transition establishes three possible relations between terms: “big-step” semantics (‘`==>`’), “small-step” semantics (‘`-->`’), and static semantics (also ‘`==>`’, overloaded), following the traditional notation of operational semantics literature [11, 13, 12]. All three relations, described by the non-terminal ⟨ relation ⟩, are ternary with the following components: the typed value-added syntactic tree to be matched against, the MSOS label expression (⟨ label-exp ⟩) enclosed between braces, and the resulting value-added syntactic tree.

A label expression is a non-empty list of field expressions (⟨ field-exp ⟩) separated by commas and enclosed between braces, where each field expression is either an index and its component or the “rest of the label.” There are

[2]Which is only decorative and should not to be confused with the MSOS label.

special metavariables for the rest of the label (⟨ rest ⟩): if it is unobservable, the metavariable '`-`' should be used, otherwise '`...`' should be used. As usual on MSOS specifications, we use `-->` and `==>` as sugar for `-{-}->` and `={-}=>`, respectively. Instead of '`...`' and '`-`', one may uses the metavariables '`X`' and '`U`' respectively. These metavariables may optionally be postfixed with a number, such as '`X1`'.

Label composition is specified using the syntax of the non-terminal ⟨ composition ⟩. It is recommended that numbered metavariables ('`X1`', '`X2`', etc.) be used on the composition to ease the understanding and to follow traditional MSOS notation. Of course, label composition only makes sense on conditional transitions.

Metavariables over sets are not declared explicitly in MSDF, but instead declared implicitly: all non-terminals of the form ⟨ dsetid ⟩ that appear in transitions are considered metavariables for their corresponding sets using this simple formation rule: given that all set identifiers are assumed to have only letters (Section 3.2), all characters that are neither letters nor the symbols '`*`' and '`+`' are stripped from the ⟨ dsetid ⟩ and the remaining is assumed to be the intended type of the metavariable.

For example: '`Exp`', '`Exp1`', and '`Exp'`' are all metavariables that range over '`Exp`', obtained by removing the characters '`1`' and '`'`', respectively, while '`Exp*1`' is a metavariable over '`Exp*`', obtained by removing the character '`1`'. The rationale for this is that, in programming language definitions, it is often the case in which a set is used on most transition rules. For example, the set of expressions, or the set of values. As in the case of module inclusions, it would probably be tedious and error prone to declare the same variables over the same modules.

As an example of unconditional transitions, label expressions and implicit metavariables let us consider the following fragment that defines the semantics of a construction that prints a computed value. It defines an unconditional transition between the typed syntactic tree '`(print Value) : Cmd`' to '`skip`', the "do-nothing" command. The semantics of the '`print`' command is defined with a write-only component by the label expression '`out' = Value`' that models the produced information '`Value`'. The rest of the record is *unobservable*, that is, any read-write component remains unchanged, no other produced information occurs on other write-only components, and read-only components will always remain the same, of course.

```
(print Value) : Cmd -{out' = Value, -}-> skip .
```

Conditional transitions have four different types of conditions, ruled by the non-terminal ⟨ cond ⟩: equality conditions, predicates, variable instantiations, and conditional transitions. Equality conditions (⟨ eq ⟩) assert the equality of two terms, such as '`first (Pids') = Int`'. A single term '`P`' is used as a predicate (⟨ pred ⟩), such as '`odd(n)`', to abbreviate the equational condition '`P = true`'. The ⟨ inst ⟩ non-terminal defines the syntax used to instantiate new metavariables. The free metavariable must be on the left-hand side of the '`:=`', as in '`Value := lookup (Id, Env)`'. Finally, ⟨ cond-step ⟩ is a conditional transition that has the same syntax as of unconditional transitions, with the exception that the type of the syntactic tree to be matched against is necessarily the *least set* that applies to the left-hand side term, that is, the smallest set in a set inclusion relation.

Let us exemplify conditional transitions with a small-step specification for the semantics of an abstract conditional construction, with syntax '`cond Exp Exp Exp`'. The semantics is the usual: the first expression is to be evaluated into a boolean value; if it is true, the second expression is evaluated, otherwise the third is. The semantics for this needs three transitions: the first states that the condition—the metavariable '`Exp`' ranging over the set '`Exp`'—must be evaluated. The term '`(cond Exp Exp1 Exp2) : Exp`' is a typed syntactic tree with the explicit type '`Exp`', while on the condition, the typed syntactic tree '`Exp`' has as implicit type the least set applicable to the metavariable, which is also '`Exp`'. The condition states that, if '`Exp`' is evaluated into '`Exp'`', then the left-hand side of the main transition evaluates to '`cond Exp' Exp1 Exp`'. The label '`...`' on the condition is the same as the main transition, meaning that any changes to read-write components are propagated to the main transition, and any produced information by write-only components is also produced by the main transition. Read-only components must always remain the same.

```
                         Exp -{...}-> Exp'
-- --------------------------------------------------------
(cond Exp Exp1 Exp2) : Exp -{...}-> cond Exp' Exp1 Exp2 .
```

In the example above, we used the fact that, in Maude, three dashes ('`---`') initiate a comment line.

Finally, two additional rules are needed for each possible outcome of '`Exp`': if it is '`tt`', the whole left-hand side evaluates to '`Exp1`', otherwise into '`Exp2`'. These transitions are unobservable.

```
(cond tt Exp1 Exp2) : Exp --> Exp1 .
(cond ff Exp1 Exp2) : Exp --> Exp2 .
```

Let us further exemplify conditional transitions by defining the same conditional construction in a big-step style, which also uses three rules, but with an additional construction, as follows:

The internal construction (not part of the main syntax of the language) '`if-choose`' has three arguments: one value and two expressions. If the value is '`tt`', then it evaluates to '`Exp2`', otherwise to '`Exp3`'. The example also shows each transition rule with a rule label.

```
Exp ::= if-choose (Value, Exp2, Exp3) .

[if-choose-tt] if-choose (tt, Exp2, Exp3) : Exp ==> Exp2 .
[if-choose-ff] if-choose (ff, Exp2, Exp3) : Exp ==> Exp3 .
```

There is a single transition for the conditional construction itself, but with three conditional transitions. The first expects the expression '`Exp`' to be evaluated into '`Value`', with label expression '`X1`'. This value, when used as a parameter in '`if-choose (Value, Exp2, Exp3)`', is expected to be evaluated, in turn, into '`Exp'`', in an unobservable manner. Finally, '`Exp'`' is expected to be evaluated into '`Value'`', with label expression '`X2`'. If those three conditions are satisfied and label expressions '`X1`' and '`X2`' are composable—recalling, the read-only components remain the same, the initial value of a read-write component on '`X2`' is the final value of the same read-write component on '`X1`', while produced information does not affect composability of labels—, then the whole left-hand side is evaluated into '`Value'`'. The label '`X1 ; X2`' on the main transition specifies that the resulting label is the composition of labels '`X1`' and '`X2`'.

```
     Exp ={X1}=> Value,
     if-choose (Value, Exp1, Exp2) ==> Exp',
     Exp' ={X2}=> Value'
[if] -- ----------------------------------------
     (cond Exp Exp1 Exp2) : Exp ={X1 ; X2}=> Value' .
```

MMT checks for source-dependent variables and reports when it finds transitions that contain variables without this property, as the following example shows:

```
(msos TEST is
 Foo .
 Bar .

 Foo ::= f (Bar, Bar) .

         Bar --> Bar'
 -- -------------------------
 f(Bar, Bar) : Foo --> Bar'' .
sosm)
```

Upon reading module '`TEST`', the following error is displayed:

```
rewrites: 19081 in 96ms cpu (101ms real) (196741 rewrites/second)
ERROR: non source-dependent variables found: Bar'' in module TEST
```

## 3.5 Built-in operations on derived and parameterized sets

The parameterized and derived sets in MSDF have several associated operations, which we describe in this Section. Each function is presented as $\mathsf{f} : \mathsf{S} \to \mathsf{S}'$: a function $\mathsf{f}$ with domain $\mathsf{S}$ and codomain $\mathsf{S}'$. If the function $\mathsf{f}$ is in mixfix format, it is converted to prefix format, where the arguments are replaced by underscores ('`_`').

Let `s`, and `s'` be any two primitive sets (that is, neither derived nor parameterized sets). Recall that, from a set `s`, the sets `s*` and `s+` are automatically derived. Furthermore, the user may create the following parameterized sets: `(s)List`, `(s)Set`, and `(s, s')Map`.

## 3.6 Sequences

`_,_` : $\mathtt{s*} \times \mathtt{s*} \to \mathtt{s*}$

The monoid binary function (with optional surrounding parentheses), with identity is '`()`'. A single element `s` is also a sequence.

## 3.7 Lists

`[_]` : $\mathtt{s*} \to$ `(s)List`

Constructs a list out of a sequence of elements.

`_in_` : `s` × `(s)List` → `Bool`
Returns *true* if the `s` is on the list `(s)List`.

`first` : `(s)List` → `s`
Returns the first element from the list `(s)List`.

`remove` : `s` × `(s)List` → `(s)List`
Creates a copy of the list `(s)List` with all copies of `s` removed.

`insert-back` : `s` × `(s)List` → `(s)List`
Inserts `s` as the last element of `(s)List`.

`insert-front` : `s` × `(s)List` → `(s)List`
Inserts `s` as the first element of `(s)List`.

`length` : `(s)List` → `Nat`
Number of elements of `(s)List`.

## 3.8 Maps

`_|->_` : `s` × `s′` → `(s,s′)Map`
Creates an entry that binds `s` to `s′`.

`_+++_` : `(s,s′)Map` × `(s,s′)Map` → `(s,s′)Map`
Disjoint union of maps.

`length` : `(s,s′)Map` → `Nat`
Number of entries on the map.

`def lookup` : `s` × `(s,s′)Map` → `Bool`
Is *true* if there exists a mapping from `s` in `(s,s′)Map`.

`lookup` : `s` × `(s,s′)Map` → `s′`
Returns the element that `s` maps to in `(s,s′)Map`.

`_/_` : `(s,s′)Map` × `(s,s′)Map` → `(s,s′)Map`
Overrides the mappings of the second `(s,s′)Map` with the ones of the first `(s,s′)Map`.

## 3.9 Sets

`size` : `(s)Set` → `Nat`
Number of elements of `(s)Set`.

`_in_` : `s` × `(s)Set` → `Bool`
Is *true* if `(s)Set` contains `s`.

`_+_` : `(s)Set` × `(s)Set` → `(s)Set`
Disjoint union of sets.

## 4 User interface

The normal operation of the *Maude MSOS Tool* is with the user inputting MSDF modules at the command prompt or by loading files using Maude's '`load`' command and by using Full Maude's own commands for rewriting, reducing, searching and model checking specifications. This Section outlines the commands that are specific to *Maude MSOS Tool*.

Currently the only command available controls the *step flag* of the compilation process. The normal operation of the MMT is with this flag *on*, since there is a need to control the rewrites on the conditions. The only case in which this step must be *off* is when there is other means of controlling these rewrites, for example, using Alberto Verdejo's strategy language for Maude [8]. The syntax of the command is either '`step flag on`' or '`step flag off`'.

## 5 A simple example

This section revisits the simple example of a language specification. This very simple programming language contains only an ML-like '`let-in-end`' construction and the '`sum`' function.

The following specification is enclosed by an MSOS module definition to be accepted in the MMT.

```
(msos SIMPLE-LANGUAGE is ... sosm)
```

The following datatype definitions declares the sets used on the specification: '`Exp`', the set of expressions, and '`Id`', the set of identifiers. We let the expressions range over identifiers and integers (the built-in set '`Int`'), the only primitive type of our simple programming language.

```
 Id .
 Exp .
 Exp ::= Int | Id .
```

We now declare the two constructions of the language.

```
 Exp ::= let Id = Int in Exp end
       | Exp sum Exp .
```

The specification of the ‘`let-in-end`’ construction requires a read-only environment for the bindings. We declare it as a map from identifiers to integers. This environment is accessed by the index ‘`env`’.

```
Env = (Id, Int)Map .
Label = { env : Env, ... } .
```

Now for the dynamic semantics. To evaluate the sum of two expressions, we first evaluate the first expression until it reaches a final value, which is specified to be an integer in this language. Then the second expression is evaluated. When final values are produced, the final value of the function itself is the mathematical sum of the two integers.

```
                 Exp1 -{...}-> Exp'1
-- -------------------------------------------
(Exp1 sum Exp2) : Exp -{...}-> Exp'1 sum Exp2 .

                 Exp2 -{...}-> Exp'2
-- -----------------------------------------
 (Int sum Exp2) : Exp -{...}-> Int sum Exp'2 .

       Int3 := Int1 + Int2
-- -----------------------------
(Int1 sum Int2) : Exp --> Int3 .
```

For the meaning of the ‘`let-in-end`’ construction, the expression is evaluated in the context of the current environment overridden with the binding provided by ‘`Id = Int`’ declaration. When the evaluation of the expression reaches the final value of an integer, the whole expression evaluates to this integer.

```
  Env' := (Id |-> Int) / Env, Exp -{env = Env', ...}-> Exp'
-- -------------------------------------------------------
(let Id = Int in Exp end) : Exp -{env = Env, ...}->
                                (let Id = Int in Exp' end) .

(let Id = Int in Int' end) : Exp --> Int' .
```

The evaluation of an identifier looks up its mapping in the environment and returns it.

```
   Int := lookup (Id, Env)
 -- --------------------------
 Id : Exp -{env = Env, -}-> Int .
```

In order to run programs with our specification we need to provide identifiers to it. We do this by creating constants '`x`' and '`y`'.

```
(msos TEST is
 see SIMPLE-LANGUAGE .

 Id ::= x | y .
sosm)
```

We may now use Full Maude's '`rewrite`' command to execute a simple program, whose argument is an MRS configuration.

```
(rewrite < (let x = 10 in
             let y = 10 in
              x sum y
             end
            end) ::: 'Exp,
            { env = void } > .)

rewrite in TEST : < ... >
result Conf :
  < 20 ::: 'Exp, { env = void } >

Bye.
```

## 6 Troubleshooting

This Section describe some shortcomings of the current implementation.

### 6.1 Limitations of MSDF

#### 6.1.1 Mixing function and subset declarations

MMT currently gets confused when, while specifying at the same time several subset inclusions and function declarations. You should always first declare the functions and then the subset declarations. For example: write '`Exp ::= foo (Exp, Exp) | Exp | Id .`' instead of '`Exp ::= Exp | Id | foo (Exp, Exp) .`'.

### 6.1.2 Maps of maps (or lists of lists, or sets of sets)

Keep in mind that you must access a parameterized set through its *equivalent set*. It is possible, for example, to create a map that maps values to other maps, such as the following example:

```
(msos FOO is
 Foo .
 Bar .

 A = (Foo, Bar) Map .
 B = (Foo, Bar) Map .
 C = (A, B) Map .

 Foo ::= a | b | c .
 Bar ::= d | e | f .
sosm)

Maude> (red ((a |-> d) |-> (a |-> d)) .)

rewrites: 955 in 40ms cpu (40ms real)
          (23875 rewrites/second)
reduce in FOO :
  a |-> d |-> a |-> d
result Entry`(A`|`B`) :
  a |-> d |-> a |-> d
```

## 6.2 Preregularity problems

The sets, subsets, and functions defined using MSDF's extended-BNF syntax are compiled into sorts, subsorts, and operators in Maude, with a one-to-one correspondence. For that reason MSDF's datatype definitions are subjected to the same restrictions that Maude's own datatype definitions have. One of these is the preregularity requirement in which a term must have one least sort. For example, the following MSDF module is non-preregular:

```
(msos NON-PREREGULAR is
 Foo .
 Bar .
 Super .

 Super ::= Foo | Bar .

 Foo ::= a .
 Bar ::= a .
sosm)
```

Currently MMT does not check whether an MSDF module satisfied the preregularity requirement or not; if it does not, it will be left to the Maude interpreter to signalize the user. For example, loading the above module into MMT and attempting to reduce the term '`a`' will generate the following two warnings from Maude:

```
Maude> (red a .)
Warning: sort declarations for operator `(`) failed
         preregularity check.
Warning: sort declarations for operator a failed
         preregularity check.
```

The second warning is expected due to the non-preregularity of '`a`'. The first needs more explanation. All sets in MSDF generate derived sets. For example, the sets '`Foo`', '`Bar`', and '`Super`' generate the derived sets '`Seq(Foo)`', '`Seq(Bar)`', '`Seq(Super)`', among others, that represents tuples of these corresponding sets. Each sequence has as identity the empty tuple, or '`()`'. Notice that this identity element is subjected to the *same* preregularity requirement of the term '`a`'. At the moment, there is no solution for this warning, but it is at the same time harmless unless two empty tuples are being used on the same term.

A related issue to the preregularity requirement is the fact that Maude is capable to specifying *context-free grammars* so, obviously, there is no support for non-context free grammars in MMT.

## 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 and José Meseguer. Modular rewriting semantics in practice. *Electronic Notes in Theoretical Computer Science*, 117:393–416, 2005.

[3] Fabricio 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] Manuel Clavel, Francisco Durán, Steven Eker, Narciso Martí-Oliet, Patrick Lincoln, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.1).* SRI International and University of Illinois at Urbana-Champaign, `http://maude.cs.uiuc.edu`, March 2004.

[5] 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 *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.

[6] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[7] Donald E. Knuth. *Selected Papers on Computer Languages.* CSLI Publications, Stanford, CA, USA, 2002.

[8] José Meseguer, Narciso Martí-Oliet, and Alberto Verdejo. Towards a strategy language for Maude. In Narciso Martí-Oliet, editor, *Proceedings of 5th International Workshop on Rewriting Logic and its Applications, WRLA 2004*, volume 117 of *Eletronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005.

[9] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

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

[11] Peter 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] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley Professional Computing. John Wiley & Sons, Chichester, England, 1992.

[13] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004. Special issue on SOS.