Journal of Universal Computer Science, vol. 10, no. 7 (2004), 789-807
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © J.UCS

A Modular Rewriting Semantics for CML

Fabricio Chalub
(Universidade Federal Fluminense, Brazil
`frosario@ic.uff.br`)

Christiano Braga
(Universidade Federal Fluminense, Brazil
`cbraga@ic.uff.br`)

Abstract: This paper presents a modular rewriting semantics (MRS) specification for Reppy's Concurrent ML (CML), based on Peter Mosses' modular structural operational semantics specification for CML. A modular rewriting semantics specification for a programming language is a rewrite theory in rewriting logic written using techniques that support the modular development of the specification in the precise sense that every module extension is *conservative*. We show that the MRS of CML can be used *to interpret* CML programs using the rewrite engine of the Maude system, a high-performance implementation of rewriting logic, and *to verify* CML programs using Maude's built-in LTL model checker. It is assumed that the reader is familiar with basic concepts of structural operational semantics and algebraic specifications.

Key Words: rewriting logic, semantics of programming languages, modularity, Concurrent ML
Category: F.3.1, F.3.2

1 Introduction

Rewriting logic is a logical framework [14] which can represent in a natural way many different logics, languages, operational formalisms and models of computation. Having different high-performance implementations [2, 5, 8], including the Maude system, RWL can be used to create powerful analysis tools for programming languages, such as JavaFAN [7], a rewriting logic-based analysis tool for Java programs. Modular rewriting semantics (MRS) [4, 16] is a novel technique for the modular specification of programming languages semantics in rewriting logic. A MRS specification is a rewrite theory in rewriting logic developed according to some techniques that supports modular definitions, that is, each module *extension* being a *conservative* one.

MRS has a close relation with Mosses' modular structural operational semantics (MSOS) [21]. This is due to the fact that structural operational semantics has a direct representation in rewriting logic [12, 3, 27, 16, 4] and that MRS and MSOS use a similar technique to achieve modularity based on the encapsulation of the semantic information. Moreover, MRS builds on MSOS insights with new techniques to achieve modularity and reuse of programming languages semantics specifications. In [16] the second author and José Meseguer propose a semantics-preserving transformation from MSOS to MRS with a formal proof of bisimulation between the models of MSOS and MRS.

Reppy's Concurrent ML (CML) [25], is an extension of Milner's Standard ML [18] with concurrency features. Both languages have formal semantics and have several implementations [11, 26, 13, 1] including several different tools, such as [23, 22].

The objective of this paper is twofold: (i) to define a MRS for CML; and (ii) to show how CML programs can be executed and model checked in rewriting logic using the MRS of CML and the Maude system. The MRS of CML is the result of the application of the semantics-preserving translation between MSOS and MRS, developed by the second author and Meseguer, to Mosses' MSOS specification of CML [19].

This paper is organized as follows. Section 2 gives the necessary background in rewriting logic and modular rewriting semantics. Section 3 presents the modular rewriting semantics of CML. Section 4 presents the execution and model checking of CML programs using Maude and the MRS of CML. We conclude this paper in Section 5 with our final remarks.

2 Rewriting Logic and Modular Rewriting Semantics

2.1 Rewriting Logic and Modularity Requirements

By a rewriting semantics for a programming language $\mathcal{L}$ we mean a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma, E, \phi, R)$, where the programs and semantic entities associated to $\mathcal{L}$ are specified by the equational theory (Σ, E), and where the operational semantics of $\mathcal{L}$ is formally specified by the rewrite rules R. The notion of *modularity* is defined in the context of an incremental specification, where syntax and corresponding semantic axioms are introduced for groups of related features. That is, modularity is a property of semantic definitions of *language extensions*. In the following paragraphs, assume that we have defined the semantics of a language fragment $\mathcal{L}_0$ by means of a rewrite theory $\mathcal{R}_{\mathcal{L}_0}$, and the semantics of a language extension $\mathcal{L}_1$, with $\mathcal{L}_0 \subseteq \mathcal{L}_1$, by means of another rewrite theory $\mathcal{R}_{\mathcal{L}_1}$.

The most basic and obvious modularity requirement is *monotonicity*: there is a theory inclusion $\mathcal{R}_{\mathcal{L}_0} \subseteq \mathcal{R}_{\mathcal{L}_1}$. Monotonicity means that we do not need to *retract* earlier semantic definitions in a language extension. Monotonicity is not easy to get. For example, standard SOS specifications are typically nonmonotonic and therefore unmodular [21]; that is, often many SOS rules for $\mathcal{L}_0$ have to be *redefined* in order to extend $\mathcal{L}_0$ to $\mathcal{L}_1$.

A second natural modularity requirement is *ground conservativity*: for any ground Σ_0-terms $t, t' \in T_{\Sigma_i,k}$ (the set of k-kinded ground Σ_i terms) we have, (i) $E_0 \vdash t = t' \Leftrightarrow E_1 \vdash t = t'$, (ii) $\mathcal{R}_{\mathcal{L}_0} \vdash t \longrightarrow t' \Leftrightarrow \mathcal{R}_{\mathcal{L}_1} \vdash t \longrightarrow t'$. Ground conservativity means that new semantic definitions *do not alter* the semantics of previous features on the previously defined language fragments. The issue then is finding *methods* ensuring that incremental rewriting semantics definitions of programming languages are *modular* in the sense of satisfying the above requirements.

MRS uses pairs, called *configurations*; the first component is the *program text*, and the second a *record* whose fields are the different *semantic entities* associated to a program's computation. We can specify configurations in Maude with the following membership equational theory (a Maude functional module importing the `RECORD` module, shown later):

```
fmod CONF is
 protecting RECORD .

 sorts Program Conf .

 op <_,_> : Program Record -> Conf [ctor] .
endfm
```

The module `CONF` is declared using the syntax `fmod`. It first includes the module `RECORD` in *protecting* mode, that is, adding no more data ("no junk") and no new equalities ("no confusion") to records. Then the sorts `Program` and `Conf` are declared using syntax `sorts`. Finally, the mixfix operator `<_,_>` is declared using syntax `op`. The `ctor` attribute specifies that this operation is a *constructor* for sort `Conf`.

The first key modularity technique is *record inheritance*, which is accomplished through pattern matching *modulo* associativity, commutativity, and identity. Features added later to a language may necessitate adding new semantic components to the record; but the axioms of older features can be given once and for all in full generality: they will apply just the same with new components in the record. The Maude specification of the equational theory of records is as follows.

```
fmod RECORD is
 sorts Index Component .
 sorts 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 `Field` is defined as a pair of `Index` and a `Component`; illegal pairs will be of kind `[Field]`. A `PreRecord` is a possibly empty (`null`) multiset of fields, formed with the union operator `_,_` which is declared to be *associative* (`assoc`), *commutative* (`comm`), and to have `null` as its *identity* (`id`). Maude will then apply all equations and rules *modulo* such equational axioms [5]. Note the conditional membership (`cmb`) defining a `Record` as an "encapsulated" `PreRecord` with no duplicated fields.

Record inheritance means that we can always consider a record with more fields as a special case of one with fewer fields. For example, a record with an environment component indexed by `env` and a store component indexed by `st` can be viewed as a special case of a record with just the environment component. Matching modulo associativity, commutativity, and identity supports record inheritance, because we can always use an extra variable `PR` of sort `PreRecord` to match *any extra fields the record may have.* For example, the function `get-env` extracting the environment component can be defined by `eq get-env(env : E:Env, PR:PreRecord) = E .` and will apply to a record with any extra fields that are matched by `PR`.

The second key modularity technique is the systematic use of *abstract interfaces.* That is, the sorts specifying key syntactic and semantic entities are *abstract sorts* such that: (i) they only specify the *abstract functions* manipulating them, that is, a given *signature*, or *interface*, of abstract sorts and functions; *no axioms* are specified about such functions *at the level of abstract sorts*; (ii) in a language specification no *concrete* syntactic or semantic sorts are ever identified with abstract sorts: they are always either specified as *subsorts* of corresponding abstract sorts, or are mapped to abstract sorts by *coercions*; it is *only at the level of such concrete sorts* that *axioms* about abstract or auxiliary functions are specified.

This means that we make no a priori ontological commitments as to the nature of the syntactic or semantic entities. It also means that since the only commitments ever made happen at the level of *concrete sorts*, one remains forever free to introduce new meaning and structure in a language extension.

Systematic use of the above two new techniques will ensure that the rewriting semantics of any language extension $\mathcal{L}_0 \subseteq \mathcal{L}_1$ is always modular, that is, that it meets the two requirements explained in Section 2.1, provided that: (i) the only rewrite rules in the theories $\mathcal{R}_{\mathcal{L}_0}$ and $\mathcal{R}_{\mathcal{L}_1}$ are semantic rules

$$\langle f(t_1, \cdots, t_n), u\rangle \longrightarrow \langle t', u'\rangle \textit{ if } C,$$

where C is the rule's condition, f, is a language feature, e.g., `if-then-else`, u and u' are record expressions and u contains a variable `PR` of sort `PreRecord` standing for unspecified additional fields and allowing the rule to match by record inheritance; (ii) the following *information hiding* discipline should be followed in

u, u', and any record expression appearing in C: besides any record syntax, only function symbols appearing in the *abstract interfaces* of some of the record's fields can appear in record expressions; any auxiliary functions defined in concrete sorts of those field's components should never be mentioned; and (iii) the semantic rules of each programming language feature f should all be defined in the *same* theory, that is, either all are in $\mathcal{R}_{\mathcal{L}_0}$ or all in $\mathcal{R}_{\mathcal{L}_1}$.

2.2 Relationship with Modular SOS

In this section we briefly discuss on the need of controlling rewriting steps in rewriting logic in order to represent the so-called "small step" specifications. For a complete discussion on the relationship with MSOS we refer to [16].

There are two main techniques for the specification of transition rules in operational semantics: structural (or *small-step*), proposed by Plotkin [24] and natural (or *big-step*), proposed by Kahn [10]. Therefore, in the context of representing an operational semantics specification in rewriting logic it is important to be able to control the *number of steps* of rewrites in the conditions of a rule. Note that in a rewrite rule

$$Q \longrightarrow Q' \ \mathit{if}\ P_1 \longrightarrow P'_1 \wedge \cdots \wedge P_n \longrightarrow P'_n$$

the rewrites $P_i \longrightarrow P'_i$ in the conditions are considerably more general: they can have zero, one, or more steps of rewriting because the rewriting relation is reflexive and transitive in rewriting logic. The point is that, by definition, in rewriting logic *all finitary computations are always derivable as sequents.* Thus, to be able to control the rewrite steps in MRS specifications the following has to be done: (i) The module `CONF` is extended to a system module (rewrite theory):

```
mod RCONF is
 extending CONF .

 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
```

(ii) Each semantic rewrite rule is of the form,

$$\{t, u\} \longrightarrow [t', u'] \ \mathit{if}\ \{v_1, w_1\} \longrightarrow [v'_1, w'_1] \wedge \cdots \wedge \{v_n, w_n\} \longrightarrow [v'_n, w'_n] \wedge C \quad (1)$$

where $n \geq 0$, and C is a (possibly empty) equational condition involving only equations and memberships. Any such application of the `step` rule exactly mimics a one-step rewrite with a rule of the form of Equation 1.

3 Modular Rewriting Semantics of CML

Standard ML (SML) is a general purpose language that has functional, imperative, and exception handling constructions among other features. SML is strongly typed with static bindings. Concurrent ML (CML) is an extension of SML with concurrency primitives added. This section presents a modular rewriting *dynamic* semantics of a significant subset of CML, based on Peter Moses' MSOS specification as described in [19]. Therefore type checking and type inference issues will not be addressed in this paper. Of course, MRS could also be used to specify the static semantics for CML following, for instance, the same approach in [19]. It is worth mentioning that the MRS of CML is correct with respect to Mosses' MSOS of CML, since our specification is produced as a result of the systematic application of the semantics-preserving translation from MSOS to MRS to Mosses' MSOS of CML.

This section is organized as follows. For three language fragments, namely, constant declarations (Section 3.1), variable assignment (Section 3.2) and thread synchronization (Section 3.3), we: (i) show the CML syntax; (ii) give the intuitive semantics for the language fragment; and (iii) present the MRS of the language fragment. (The complete specification can be obtained in full at the address `http://www.ic.uff.br/~cbraga/losd/specs/cml.maude` and also gives semantics for pattern matching and exception handling constructions.)

3.1 Declarations

In SML, constant declarations are written as **let in end** expressions, such as:

```
let val x = 1
in
    x + 1
end
```

The informal meaning for this simple example is that the expression `x + 1` is evaluated considering the *environment* determined by the *binding* from `x` to `1`. Moreover, if the identifiers `x` is also declared in an outermost **let**, the innermost declaration prevail. The environment where `x + 1` is evaluated into also has the declarations made "outside" the **let in end** expression above, that are *not* subsumed by `x`.

Let us know turn to the MRS for constant declarations. First we apply the *record inheritance* technique. The declarations environment is represented by the sort `Env`. In order to place this component in the record, we use the index `env`, a constructor of sort `Index`. We also add a membership axiom stating that an environment associated to the index `env` (through the "`:`" operator) is a `Field`, that is, part of the record structure. These two declarations are written in Maude as follows.

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

Next we apply the *abstract interface* technique by declaring functions that deal with bindings of a given environment. Two such functions are `find` and `override` declared with following signature in Maude:

```
op find : Env Ide -> [BVal] .
op override : Env Ide BVal -> Env .
```

The function `find` is declared as a *partial* function since it may return values on the *kind* `[BVal]`, that is, a certain identifier may not be present in a given environment. The function `find` returns the value bound to the identifier in the environment otherwise. The `override` function also behaves as expected and returns a new environment where the binding containing the identifier is overwritten. The environment is extended otherwise.

The following Maude rule specifies the semantics of a **let** expression with its declarations part already evaluated.

```
crl { let b:Env in e:Exp end, { (env : rho:Env), pr:PreRecord} } =>
    [ let b:Env in e':Exp end, { (env : rho:Env), pr':PreRecord } ]
 if rho':Env := override-env (rho:Env, b:Env)  /\
    { e:Exp, { (env : rho':Env), pr:PreRecord} } =>
    [ e':Exp, { (env : rho':Env), pr':PreRecord } ] .
```

This rule specifies that the expression `e:Exp` is to be evaluated considering bindings in `b:Env`. The **let** expression may be nested deep inside into the program text, so possibly there is already an outer binding environment, represented in the rule as the variable `rho:Env`. Notice that the rest of the record is captured by the variable `pr:PreRecord`. This is an example of the *record inheritance* technique for modular specification, that is, when a new language fragment is added there will be no need for this rule to be changed at all.

In order to evaluate `e:Exp` it is necessary to *override* the current environment `rho:Env` with the new bindings defined by `b:Env`. The new environment is represented by `rho':Env` and is the result of the application of the function `override-env`. This is an *abstract function*, that is, at this point no assumptions are made about how the function is implemented and the *structure* of the environment. For example, in a further extension one might want to create a new implementation of the environment that keeps track of how many times a particular identifier was looked up. For that one would need to change the concrete representation of bindings and add a counter to each binding, which would not force any change to the rule above. This rule is written once and for all, and as Mosses properly states, is definitive [20]. However it is not definitive only due to the record inheritance but *also* due to the use of abstract interfaces [16].

The evaluation of `e:Exp` with the new environment `rho':Env` rewrites to `e':Exp` and possible modifications to the rest of the record specified by `pr':PreRecord`. However the *original* environment is kept on the right hand side of the rule, specifying that the environment does not allow *side effects*. This evaluation proceeds until the expression turns into a computed value. At this point the whole **let** expression is replaced by the computed value.

3.2 Variable Assignment

SML supports imperative constructions while maintaining the functional characteristics of the bindings by associating values to identifiers through a *reference* [24]. Referenced values are created with the `ref` construction and can be bound to identifiers (for example `val x = ref 0`). One can access the original value by dereferencing a variable using the dereference operator `!`, as in `!x`. The assignment of a new value to a variable is done with the `:=` construction, which expects that the left hand side is bound to a reference value. The result of an assignment is simply the empty tuple. An example that explores these three constructions is as follows.

```
let val x = ref 0
in
    x := ! x + 1
end
```

In the MRS of CML references live in a new record component, the store. The store associates *memory locations* with values. As before, no assumption about the structure of memory locations and its associated storage is made. We then use both components, the environment and store, by binding identifiers to memory locations in the memory that in turn are associated with values in the store. In this way, although the bindings from identifiers to memory locations can not change, the values referenced by those memory locations can.

Similarly to the environment, we declare an index operator and a membership equation, therefore applying the record inheritance technique. The sort `SVal` represents the set of "storable" values. The index `st` holds an element of sort `Store` and the specification that a pair `st : S:Store` is of sort `Field`, that is, part of the record structure, is given by the membership equation below.

```
op st : -> Index [ctor] .
mb st : S:Store : Field .
```

Abstract functions on the data type `Store` are also defined such that no commitments are made with respect to the internal structure of stores. The functions `lookup` and `update` behave as expected. The former is a partial function that returns the value assigned to a given location in a given store, if the given location exists. The latter updates the given store in the given location with a given value or extends the store otherwise.

```
op lookup : Store Loc -> [SVal] .
op update : Store Loc SVal -> Store .
```

The evaluation of an assignment begins with the evaluation of the expression `E1:Exp := E2:Exp`. (The rule is not shown here.) First `E1:Exp` is evaluated until it becomes a memory location; then `E2:Exp` is evaluated until it becomes a computed value. The following Maude rule formalizes the assignment of a computed value to a memory location.

```
crl { l:Loc := v:Value, {(st : sigma:Store), pr:PreRecord} } =>
    [ tuple(), {(st : sigma':Store), pr:PreRecord} ]
  if sigma':Store := update (sigma:Store, l:Loc, v:Value) .
```

The assignment of a value (`v:Value`) to a memory location (`l:Loc`) modifies the store component from its original value (`sigma:Store`) to its new value (`sigma':Store`) where the old value associated with the memory location is replaced with the new value. The remaining of the record (`pr:PreRecord`) is left unchanged. The entire assignment is then rewritten to the empty tuple (`tuple()`).

3.3 Concurrency Primitives

Reppy's Concurrent ML [25] is an extension of SML with concurrency primitives. This section shows the MRS specification for thread creation and synchronization constructions with a discussion regarding the latter.

New threads are created with the `spawn` construction. Sending and receiving values is done using the `send` and `recv` constructions respectively. The communication between threads is done through channels that are created with the `channel` construction. In the example below two threads are created: one, bound to the identifier `x`, sends a value through the channel bound to the identifier `c` and another, bound to the identifier `y`, that expects to receive a value from the same channel `c`. The communication between threads is synchronous, that is, both threads *block* on send and receive.

```
let val c = channel () ;
        x = (fn () => send c, 10) ;
        y = (fn () => recv c)
in spawn x ; spawn y
end
```

The MRS specification for concurrency primitives is based on a simplified version of Reppy's CML semantics. It introduces two new components into the record, of the following sorts: `Acts`, with record index `ac`, that captures the signal of creation of a new thread, and `Pids`, with record index `pids`, that keeps the ids of the existing threads. Each thread is a sequence of expressions with an associated *thread id*. A new thread id is created using the `new-pid` *abstract function*. The inclusion of a pid into the list of current pids is done with the `add-pid` abstract function. The `spawn` construction is defined by the following rule.

```
crl { spawn v:Value, {(pids : ps:Pids), (ac : a:Acts), pr:PreRecord} }
    => [ pi:Pid, {(pids : add-pid(ps:Pids, pi:Pid)),
         (ac : concat(a:Acts, act(prc(pi:Pid, (v:Value tuple())))),
          pr:PreRecord} ] if pi:Pid := new-pid (ps:Pids) .
```

This rule specifies the spawning of an *anonymous* function. The variable `v:Value` contains a *closure*, a function abstraction structured as a pair, with the following components: (i) another pair, composed by a function body and its formal parameters; and (ii) the environment available to the function declaration. The closure value is appended to the element of sort `Acts` bound to the *index* `ac`. When a new thread is inserted on the `ac` component, the rule below detects this and moves the thread from the `ac` component to the pool of threads.

```
crl { prc(pi1:Pid, e1:Exp), {(ac : a:Acts), pr:PreRecord} } =>
   [ (prc(pi1:Pid, e'1:Exp) || p:Procs), {(ac : a:Acts), pr':PreRecord} ]
if { e1:Exp,  {(ac : a:Acts), pr:PreRecord} } =>
   [ e'1:Exp, {(ac : a':Acts), pr':PreRecord} ] /\
   act(p:Procs) := last(a':Acts) .
```

The semantics for thread evaluation is implemented by means of nondeterministic choice, thus specifying an *interleaving* semantics. Initially the entire program is contained in a single thread. As threads are created, they join a pool of threads which has the *associative-commutative* operator `||` as constructor. As the computation proceeds, each thread is nondeterministically selected for evaluation from the thread pool.

The following rule nondeterministically selects one thread from the pool of pids to execute. Note that only one rule is necessary because of the *associative-commutative* matching. (This simplification is not present in Mosses' MSOS specification of CML, which uses two rules for the nondeterministic choice.)

```
crl { p1:Proc || PS2:Procs, r:Record } =>
    [ PS1:Procs || PS2:Procs, r':Record ]
 if { p1:Proc, r:Record } => [ PS1:Procs, r':Record ] .
```

Synchronization is achieved with two new components, of the following sorts: `Chans`, whose elements are in the `chans` index which keeps track of the created channels, and `Offers`, whose elements are in the `offer` index that holds the requests for synchronization that a thread emits. Thus, when a thread wishes to send or receive information on a channel, it creates an offer and stores it in the `offer` index. In order for two threads to synchronize offers must *agree*. The predicate `agree` below only returns true when the offers are: `snd(ch:Chan, v:Value)` and `rcv(ch:Chan)`. The partial function `agree-value` returns `v:Value` when the offers agree. The following signature in Maude declares the `offer` index, the `Offer` constructors `snd` and `rcv`, the `agree` predicate, and the `agree-value` partial function.

```
op offer : -> Index .
mb offer : o:Offers : Field .
```

```
op snd : Chan Value -> Offer .
op rcv : Chan -> Offer .
op agree : Offer Offer -> Bool .
op agree-value : Offer Offer -> [Value] .
```

The following rule specifies the creation of new channels by adding them to the `ch` component that holds elements of sort `Chan`.

```
crl { channel tuple(), {(chans : c:Chans), pr:PreRecord} } =>
    [ ch:Chan, {(chans : add-chn(c:Chans, ch:Chan)), pr:PreRecord} ]
  if ch:Chan := new-chn (c:Chans) .
```

When a thread wishes to send a value `v:Value` to a channel `ch:Chan` it signals a *send offer* which is added to the `offer` component.

```
rl { send tuple(ch:Chan,v:Value),{(offer : o:Offers),pr:PreRecord} } =>
   [ tuple(), {(offer : set snd(ch:Chan, v:Value)), pr:PreRecord} ] .
```

The `recv` construction needed special treatment in the Maude implementation of its MRS specification. Therefore let us first informally explain its MSOS semantics. When a thread p wishes to receive a value from a given channel, it places an offer of the form `rcv(ch:Chan,v:Value)` in the `offer` index. The variable `v:Value` is then substituted by a computed value when p synchronizes with another thread. The substitution of `v:Value` is accomplished in MSOS through unification. However unification is not available in Maude 2.1, the latest alpha release at this time.

The MRS rule for `recv` specifies the following. The term `recv ch:Chan` rewrites to a *placeholder* `recv-ph(ch:Chan)` that will be *substituted* by the received value when the thread that has signaled `recv ch:Chan` synchronizes with another thread through channel `ch:Chan`.

```
rl { recv ch:Chan, {(offer : o:Offers), pr:PreRecord} } =>
   [ recv-ph (ch:Chan), {(offer : set rcv(ch:Chan)), pr:PreRecord} ] .
```

When two threads signal offers that *agree*, they can be evaluated at the same time, as specified by the following Maude rule.

```
 crl { p1:Proc || PS:Procs, {(offer : o:Offers), pr:PreRecord} } =>
     [ p'1:Proc || update-recv (PS':Procs, v:Value),
                              {(offer : o:Offers), pr:PreRecord} ]
   if { p1:Proc, {(offer : o:Offers), pr:PreRecord} } =>
      [ p'1:Proc, {(offer : set o1:Offer), pr:PreRecord} ] /\
      { PS:Procs, {(offer : o:Offers), pr:PreRecord} } =>
      [ PS':Procs, {(offer : set o2), pr:PreRecord} ] /\
      agree(o1:Offer, o2:Offer) /\
      v:Value := agree-value (o1:Offer, o2:Offer) .
```

There are two important issues to be explained in the rule above: (i) the application of function `update-recv` that performs the above mentioned substitution, and (ii) the predicate `agree` and the function `agree-value`. We follow this order and begin with the application of function `update-recv`. During synchronization the substitution of the placeholder `recv-ph ch` by a value is done by the function `update-recv`, which is actually a *meta-function.* Before discussing `update-recv`, let us briefly comment on meta-functions in rewriting logic and their implementation in Maude.

Rewriting logic is reflective [14], that is, a rewrite theory $\mathcal{R}$ can be meta-represented as a term $\overline{\mathcal{R}}$, defined according to an *universal* rewrite theory that can represent all other theories including itself. This is the intuition behind the so-called *reflective tower* in rewriting logic which gives formal support for the definition of meta-applications in Maude, such as execution environments for specification languages (e.g. [3]). Such applications are implemented in Maude as meta-functions (and meta-rules) that rely on the so-called descent functions: primitive meta-functions in the Maude system. One such descent function is `upTerm` which receives a term t and returns $\overline{t}$ the meta-representation of t. The `downTerm` function produces the object representation t of a given meta-represented term $\overline{t}$.

Let us return to `update-recv`. This function is defined in terms of the above mentioned `upTerm` and `downTerm` descent functions. As a meta-function, `update-recv` is defined inductively on the structure of terms in the *Maude* language, and *not* inductively on the *CML* syntax as would be the case if `update-recv` was to be defined as a function at the object level.

A definition of `update-recv` at the object level would imply an axiomatization by structural induction on the CML syntax. However, this approach is *not* modular: any extension to CML would require an extension to this function. Defining `update-recv` as a meta-function *preserves* the modularity of the specification, as opposed to an object-level definition on the CML syntax, because, as we mentioned before, it is defined inductively on the structure of *terms* in the Maude language. Therefore, an extension to the CML syntax does *not* imply an extension to `update-recv`.

We continue the explanation of the synchronization rule discussing the predicate `agree` and the function `agree-value`. By the definition of `agree`, the synchronization rule above is only applied when `p1:Proc` is currently executing a `send` and `p2:Proc` is currently executing a `recv`. At this point `p1:Proc` will write an offer on its `offer` component that will match the offer that `p2:Proc` also writes on its own `offer` component. After that, both threads evaluate one step of computation forward at the same time (compare with the interleaving rule above) and update the `p2:Proc` thread by substituting the placeholder value with a computed value.

With the rules so far, the means for a thread to block until its offer does not match with another offer from another thread are not specified. This is accomplished by the following rule.

```
crl { cml P:Procs, {(offer : offer-id), pr} } =>
    [ cml P':Procs, {(offer : offer-id), pr'} ]
 if { P:Procs,  {(offer : offer-id), pr} } =>
    [ P':Procs, {(offer : offer-id), pr'} ] .
```

This rule specifies that the multiset of threads may evolve when the `offer` index is empty (`offer-id`). This captures both situations: when two threads synchronize, or no thread at all requires synchronization.

4 Executing and Verifying CML Programs with MRS

4.1 Executing the Factorial Function

We used two different implementations of the factorial function: the traditional recursive (functional) and a iterative (imperative) version. The execution of the factorial of 300 took 198.57 seconds with 9,512,075 rewrites and 7.5 seconds with 351,836 rewrites, respectively.

All the rules in the MRS of CML described in Section 3 are written in the so-called *small-step* form. Therefore, all the intermediate steps in a given computation are observable. This is not the case in the so-called *big-step* operational semantics. In big-step specifications, intermediate steps are not observable, which, of course, reduces the number of steps in computations. For this reason, we have developed a big-step, equational version of the MRS of SML, which can be found at `http://www.ic.uff.br/~cbraga/losd/specs/sml.maude`. The equational version of the MRS of SML was systematically produced from the MSOS of CML: inference rules are represented by conditional equations, transitions in the premises of inference rules are represented by the so-called *matching equations* [5], and, of course, the right hand side of equations are always values. In the equational version of the MRS of SML, the execution of the factorial of 300 took 0.45 seconds with 23,122 rewrites and 0.52 seconds with 33,511 rewrites, for the recursive and iterative cases, respectively. It is interesting to note that these execution times are comparable to the Moscow ML [26] implementation of SML, using Joe Hurd's **bignum** library[1].

Table 1 summarizes these execution times. Column *eq* shows the execution times and the number of rewrites of the equational, big-step specification of SML. Column *rl* shows the execution times and the number of rewrites of the rule-based, small-step specification of CML. Column *mosml* shows the execution times of the Moscow ML interpreter. The recursive factorial function is represented by row *fat-rec* and the iterative version by row *fat-nrec*. Times are in seconds.

[1] `http://www.cl.cam.ac.uk/~jeh1004/software/bignum.htmlx`

	rl		eq		mosml
	time	rewrites	time	rewrites	time
fat-rec	198.57	9512075	0.45	23122	3.30
fat-nrec	7.50	351836	0.52	33511	0.54

Table 1: Executing the factorial function in Maude and Moscow ML

4.2 Executing a concurrent program

In this section we demonstrate the execution of a concurrent, nondeterministic program. The following code consists of one single thread that spawns two additonal threads that attempt to synchronize with the main thread, through the pair `send`–`recv`. Two of them will eventually *rendezvous*, that is, a *send* event will *agree* with a *receive* event, and either the value 10 or 11 will be transferred from one thread to another through channel `c`.

```
cml let val c = channel ();
        f = fn () => send (c, 10); g = fn () => send (c, 11);
    in spawn f; spawn g; recv c; end
```

Using Maude's `search` command, we find two final states, starting from the initial state above. Remember that a thread consists of a program id and an expression that will possibly be evaluated to a final value. On the first final state, the main thread, with pid 1, contains the value 10 and on the second final state, the main thread contains the value 11. On both states the other thread that didn't synchronize, that is, with pids 3 and 2, respectively, is stopped on the `send` expression. The thread that executed the successful `send` ended up with the empty tuple (`tuple()`) as a final value. We used ... to hide unnecessary details from the code below.

```
search in EXPR-TEST :
 < cml prc(pid(1),
      let val pat(c) = channel .() seq
              pat(f) = (fn p() => send .(c,rat(10))) seq
              pat(g) = (fn p() => send .(c,rat(11)))
    in spawn f ; spawn g ; recv c end), init >
=>! C:Conf .

Solution 1 (state 163)
states: 165  rewrites: 67305 in 3950ms cpu (4030ms real) (17039
    rewrites/second)
C:Conf --> < cml (prc(pid(1), rat(10)) ||
                  prc(pid(2), tuple()) ||
                  prc(pid(3),
                     let < ... > in
                       let < mt-env > in
                          send tuple(chn(1),rat(11))
                       end
                     end)), {...} >
```

```
Solution 2 (state 164)
states: 165  rewrites: 67490 in 3970ms cpu (4044ms real) (17000
    rewrites/second)
C:Conf --> < cml (prc(pid(1), rat(11)) ||
                  prc(pid(2),
                     let < ... > in
                       let < mt-env > in
                         send tuple(chn(1),rat(10))
                       end
                     end)              ||
                  prc(pid(3), tuple())), {...} >

No more solutions.
states: 165  rewrites: 67490 in 3970ms cpu (4046ms real) (17000
    rewrites/second)
```

4.3 Model Checking Dekker's Algorithm for Mutual Exclusion

Maude comes with a built-in LTL model checker [6]. In what follows it is exemplified the use of the MRS of CML together with the Maude model checker to verify Dekker's algorithm [6], one of the earliest correct solutions to the mutual exclusion problem. The algorithm assumes threads that execute concurrently on a shared memory machine and communicate with each other through shared variables. There are two threads, `p1` and `p2`. Thread `p1` sets a Boolean variable `c1` to 1 to indicate that it wishes to enter its critical section. Thread `p2` does the same with variable `c2`. If one thread, after setting its variable to 1 finds that the variable of its competitor is 0, then it enters its critical section right away. In the case of a tie (both variables to 1) the tie is broken using a variable `turn` that takes values in $\{1, 2\}$. See `http://www.ic.uff.br/~cbraga/losd/specs/cml.maude` for Dekker's algorithm in our CML syntax.

The program text consists of an initial thread that spawns two threads; each thread in turn loops entering and leaving the critical zone. While inside the critical zone each thread assigns 1 and 0 to the variables `cz1` and `cz2`. Therefore, if the implementation is correct, the store configuration with both variables set to 1 will never occur.

Let φ_{mv} be a linear temporal logic proposition that states that the mutual exclusion property was violated, that is, both memory regions related to `cz1` and `cz2` have the value 1. Thus, it must be proved that $\Box\neg\varphi_{mv}$ is true. The following module specifies φ_{mv} as the operator `mutex-violation` of sort `Prop` and the proposition we are trying to prove is represented by the formula `[]~mutex-violation`.

```
mod CHECK is
 including CONCURRENCY-TEST .
 including MODEL-CHECKER .

 subsort Conf < State .

 op mutex-violation : -> Prop .
 eq < P:Program,{(st : <[[loc(1),rat(1)]] [[loc(2),rat(1)]] C:CStore>),
      PR:PreRecord } > |= mutex-violation = true .
endm
```

The result, shown below, means that the configuration `mutex-violation` will never occur, as expected. The computer used was an Pentium IV 2.4 GHz with 512 MB RAM.

```
reduce modelCheck(dekker, []~ mutex-violation) .
rewrites: 58380093 in 2315950ms cpu (2362140ms real)
                                    (25207 rewrites/second)
result Bool: (true).Bool
```

Other checks are possible. For example: consider the proposition `competing`, meaning that both threads are competing for the critical zone. This is true when both memory regions referenced by the variables `c1` (location 5) and `c2` (location 6) are 1, as in:

```
op competing : -> Prop [ctor] .

eq < P:Program, {(st : <[[loc(5),rat(1)]] [[loc(6),rat(1)]] C:CStore>),
     PR:PreRecord } > |= competing = true .
```

Consider also the proposition `turn(i)` which is true when the turn is with the thread identified by `i`. Recall that `turn` selects which thread is allowed to enter the critical zone.

```
op turn : Int -> Prop [ctor] .

eq < P:Program, { (st : < [[loc(7),rat(i:Int)]] C:CStore >),
       PR:PreRecord } > |= turn(i:Int) = true .
```

Finally, consider the following LTL formula: $\phi = \Box(competing \rightarrow \Box turn(1))$, which can be understood as *it is always true that when both threads are competing, the turn will always be with thread 1.* When model checking this formula, Maude produces a counter-example, after 6.9 seconds and 229,112 rewrites, with 33,204 rewrites per second. The counterexample shows all paths of computations in which ϕ is not true. For example, one such computation is:

```
{< ...,{(env : < mt-env >),(st : < [[loc(1),rat(0)]] [[loc(2),rat(0)]]
[[loc(3),rat(0)]] [[loc(4),rat(0)]] [[loc(5),rat(1)]] [[loc(6),rat(1)]]
[[loc(7),rat(2)]] >),(val : < mt-val >),(pids : < pval[pid(1)] x
pval[pid(2)] x pval[pid(3)] >),(ac : < mt-ac >),tr : < mt-tr>} >,'step}
```

where one can see the memory location 7 containing the value 2, indicating that the turn is now with thread 2.

Even though the model checking of ϕ took only a few seconds, the model checking of φ_{mv} took almost forty minutes for a rather simple query and the reason for that is the same as for the high execution times shown in Section 4.1: the large number of states produced by a rule-based specification.

5 Final remarks

In this paper we have shown how CML programs can be executed and formally verified within the Maude system using the MRS of CML, a modular specification in rewriting logic for a significant subset of CML.

From this experiment it follows that the right balance between equational and rule-based axiomatizations should be pursued, following the lines of [15], which are already being followed in [17, 7]. Roughly speaking, the sequential fragment should be equationaly specified and the concurrent fragment should be rule-based. In this way the state space is restricted to concurrency-related computations, interesting for reasoning with model checking techniques.

Moreover, model checking of CML programs with conditional rewrites representing the transition rules' premises proved somewhat problematic since rewrites in the conditions are assumed "scratch pad rewrites" in RWL [14]. Thus, states that exist only in the conditions can not be specified in the query (LTL formula) to the model checker. The current MRS specification for CML only allows queries about the full program text, not its parts. With this limitation queries to the model checker must be made by observing changes to mutable components, such as the store, or by exploiting some property that involves the entire program text, and not some part.

We are currently working on a *new* CML specification which should produce several benefits when compared to the current, rule-based, one, by: (i) exploring *true concurrency*, due to the congruence rule in RWL calculus, as opposed to the interleaving model of the current, rule-based, specification; (ii) specifying the sequential fragment of CML equationaly and the concurrent part with rules, following the taxonomy proposed in [15]. This approach should lead to a transition system with fewer states, due to the use of equations to specify the sequential part, therefore making model checking faster. It is also believed that this technique should shorten execution time; (iii) move towards a *reduction semantics* [9] together with a continuation-passing style [7] in order to avoid conditional rewrites representing transition rules' premises.

Acknowledgements

The authors would like to thank the anonymous referees for their comments and suggestions, to Peter Mosses for comments on our specification. Braga acknowledges partial support from CNPq under process 300294/2003-4, and PROPP/UFF, and Chalub to EPGE-FGV.

References

1. P. Bertelsen, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, N. Rothwell, P. Sestoft, M. Tofte, and D. N. Turner. ML Kit. `http://www.itu.dk/research/`

mlkit/.
2. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. Elan from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
3. C. 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`.
4. C. Braga and J. Meseguer. Modular rewriting semantics in practice. In N. Martí-Oliet, editor, *Proceedings of 5th International Workshop on Rewriting Logic and its Applications, WRLA 2004*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004. To appear.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.
6. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
7. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *CAV'04, Proc. 16th Intl. Conf. on Computer Aided Verification, Boston, USA*, Lecture Notes in Computer Science. Springer, 2004. To appear.
8. K. Futatsugi and R. Diaconescu. Cafeobj report. *World Scientific, AMAST Series*, 1998.
9. J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, Cambridge, MA, USA, 1996.
10. G. Kahn. Natural semantics. Report 601, Inria, Institut national de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt B.P.105 78153 Le Chesnay Cedex Fance, February 1987.
11. B. Laboratories, P. University, Y. University, and A. Research. Standard ML of New Jersey. `http://www.smlnj.org/`.
12. N. Martí-Oliet and J. Meseguer. *Handbook of Philosophical Logic*, volume 61, chapter Rewriting Logic as a Logical and Semantic Framework. Kluwer Academic Publishers, second edition, 2001. `http://maude.cs.uiuc.edu/papers`.
13. D. Matthews. Poly/ML. `http://www.polyml.org/`.
14. J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.
15. J. Meseguer. Software specification and verification in rewriting logic. In *Models, Algebras and Logic of Engineering Software, NATO Advanced Study Institute, July 30 – August 11, 2002*, pages 133–193. IOS Press, 2003.
16. J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. In *AMAST'04, Proc. 10th Intl. Conf. on Algebraic Methodology and Software Technology, Sterling, UK*, Lecture Notes in Computer Science. Springer, 2004. To appear.
17. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer-Verlag, 2003.
18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (Revised).* MIT Press, 1997.
19. P. D. Mosses. Fundamental concepts and formal semantics of programming languages – an introductory course. Lecture notes, University of Aarthus, Denmark, 2002. `http://wiki.daimi.au.dk/dSprogSem-01/dSprogSem-01.wiki`.
20. P. D. Mosses. Definitive semantics. Lecture notes, Warsaw University, 2003. `http://www.mimuw.edu.pl/~mosses/DS-03`.

21. P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004. To appear.
22. U. of Cambridge Computer Laboratory. HOL, automated proof system for higher order logic. `http://hol.sourceforge.net/`.
23. L. Paulson and T. Nipkow. Isabelle, a generic theorem proving environment. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.
24. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN - 19, Computer Science Department, Aarhus University, 1981.
25. J. Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, June 1992. Technical Report TR 92-1285.
26. P. Sestoft. Moscow ML. `http://www.itu.dk/research/mlkit/`.
27. J. A. Verdejo. *Maude como um marco semântico ejecutable*. PhD thesis, Universidad Complutense Madrid, 2003.

All URLs in this paper are valid as of June 29th, 2004.