Modular Specification and Analysis of Distributed Algorithms

Fabricio Chalub and Christiano Braga

{fchalub,cbraga}@ic.uff.br

Universidade Federal Fluminense

Abstract

Modularity is an important *engineering* property that specifications should have, to allow specifications to grow larger and to modularly reason about them, when possible. These aspects fall quite appropriately in the context of the specification of distributed algorithms, where environment issues should be abstracted and the specification should focus on the problem that the algorithm claims to solve. Moreover, to reuse specifications, in a semantics preserving manner, is indeed a desirable feature. This paper proposes the use of modular structural operational semantics (MSOS) as a semantic framework for such specifications and advocates the use of the Maude MSOS Tool to support the execution and analysis of the MSOS specifications of distributed algorithms.

1 Introduction

Modularity, a quite desirable pragmatic property that specifications, and software artifacts in general, should have, means that a large specification may be built from the composition of several smaller specifications, or modules, and that the large specification preserves the semantics of its composing modules. In the context of distributed algorithms, we quote Lynch [20, page 6], one of the main references for distributed algorithms, perhaps the most cited one, in the literature: *"We have tried to make our presentations as* modular *as possible by composing algorithms to obtain other algorithms, by developing algorithms using levels of abstraction, and by transforming algorithms for one model into algorithms for other models. This helps greatly to reduce the complexity of the ideas and allow us to accomplish more with less work. The same kinds of modularity can serve the same purposes in practical distributed system design."* Indeed the modular presentation of the algorithms in Lynch's book is important to ease the understanding and reasoning about the algorithms. The execution environment, for instance, is abstracted and *scheduling* strategies are *assumed* fair, when necessary.

Lynch does not commit her presentation with a particular framework willing to focus on the aspects intrinsic to the algorithms themselves. However, when building *tool support* for the specification and analysis for such algorithms and systems one needs to make such commitment. Plotkin's structural operational semantics (SOS) [36] is a well known framework for the specification of programming languages semantics [37, 30] and also for *concurrent* systems [28, 29]. Therefore, an appropriate choice as a semantic framework for the specification of distributed algorithms if not for the lack of support for modular specifications. Nevertheless, the modularity problem in SOS specifications was solved recently by Mosses with a new framework named *modular* structural operational semantics (MSOS) [32] that essentially generalizes labeled transition systems by structuring the labels as *extensible* records. Such generalization allows for the description of highly modular specifications that may be *conservatively* extended, thus presenting MSOS as an interesting candidate as a semantic framework for distributed algorithms and systems.

We propose MSOS as a semantic framework for the specification of distributed algorithms and systems and advocate the use of *Maude MSOS Tool* (MMT) [5] as an adequate support for the execution and analysis of this class of specifications. *Maude MSOS Tool* is a *formal* tool, implemented as a realization of a semantics preserving mapping from MSOS to rewriting logic (RWL) [24] on top of the Maude [9] system, a high-performance implementation of RWL. Rewriting logic is a *reflective* logical and semantic framework quite suitable for the specification of formal tools [10, 8] due to reflection. When a logic or specification language, for instance, is mapped to RWL, techniques and tools developed for RWL also become available to such logic or specification language. Maude, as a realization of RWL with support to reflection through metaprogramming, is a natural candidate as an implementation platform. Thus, with MMT, analysis tools developed for Maude may be applied to MSOS specifications, including state search and model checking.

In this paper we exemplify how modular operational semantics specifications of distributed algorithms may be described using the modular specification definition formalism (MSDF), MMT's specification language, and how such modular specifications may be analyzed, taking advantage of its modularization, using Maude's built-in tools. For instance, different process scheduling strategies may be chosen while analyzing an algorithm without changing its specification. The application of user-defined tools, in Maude, is also proposed.

The rest of this paper is organized as follows. Section 2 gives a simple example specification to introduce MSDF notation and basic execution and analysis facilities in MMT. In Section 3 we present MMT as a formal tool using rewriting logic. Section 4 gives the specification of two distributed algorithms from Lynch's book using our approach, illustrating how distributed algorithms and systems my be *formally* and *modularly* specified and analyzed with MMT. Section 5 concludes this paper with related work and some final remarks.

2 MSDF and MMT

By modularity in specifications we mean that an existing specification $\mathcal{S}$ may be *extended* by an specification $\mathcal{S}'$ such that the meaning of $\mathcal{S}$ is not changed by $\mathcal{S}'$. In algebraic terms, $\mathcal{S}'$ extends $\mathcal{S}$ without adding *confusion* [15] to $\mathcal{S}$.

Modular structural operational semantics (MSOS) is a dialect of structural operational semantics (SOS) that solves the modularity problem left open by Plotkin in his seminal lecture notes [36] by, given an SOS specification $\mathcal{S}$: i) separating syntactic and semantic components placing the latter in the configurations and the former in the labels of the transition rules of $\mathcal{S}$; ii) structuring the labels of the transition rules of $\mathcal{S}$ as *records*, such that when new rules are added by an extension $\mathcal{S}'$ of $\mathcal{S}$, rules in $\mathcal{S}$ do not have to be retracted and the new rules in $\mathcal{S}'$ simply range over a new index in the label structure. The model of an MSOS specification is a generalized transition system (GTS), which is a transition system whose transition labels are understood as *arrows* of a category and adjacent labels in computations are required to be composable. Formally a GTS [32] is a quadruple $(\Gamma, \mathbb{A}, \rightarrow, T)$ where Γ is the configuration component for value-added syntax-trees, $\mathbb{A}$ is a category with arrows A, $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is the transition relation, and $T \subseteq \Gamma$ is the set of terminal states such that $(\Gamma, A, \rightarrow, T)$ is a labeled terminal transition system. Computation requires that whenever a transition with label α is followed by a transition labeled α', then α and α' are composable in $\mathbb{A}$. We refer the interested reader to [32] for a comprehensive presentation of MSOS. To achieve our claims, set in Section 1, we will focus on MSOS specification descriptions more than their models as GTS.

Our specifications will be written using a concrete syntax to MSOS, named modular specification definition formalism (MSDF). MSDF has three main types of declarations: i) those that involve the *syntax* of the configurations of the GTS; ii) the contents of the *labels*; and iii) the transitions themselves. We are going to present MSDF notation by means of an example specification of a simple *thread game*, where a global, shared, variable (‘`sh`’) is updated by two different threads in different ways: thread 0 increments ‘`sh`’ while it is less than a number, say 10 and, in parallel, thread 1 decrements ‘`sh`’ while it is greater than 0.

MSDF modules are written with the syntax ‘`msos` m `is` d `sosm`’, where m is the module’s name, and d are the declarations. In this case, the module name is ‘`THREAD-GAME`’.

MSDF uses the notion of *sets*, *parameterized sets*, and *functions* to define the syntax of the system being specified. In this specification we initially need a set of processes, named ‘`Proc`’. A single process is represented by a function from integers to ‘`Proc`’, named ‘`prc`’ with one single argument, an integer representing its identifier (or “pid”). In our *language-oriented* this construction is declared using BNF meta-notation.

MSOS transitions use their *labels* to carry semantic information. Each label is structured as a record, whose components can be of three different types: i) read-only components model information that never changes throughout the computation, ii) read-write components model information that may change during a transition, iii) while write-only components model information that is *produced* by a transition.

As mentioned before, information in MSDF components are accessed through *indices*. A read-write component in MSDF is written with a pair of indices (a unprimed and a primed one) that represent, respectively, the information present at the beginning and at the end of a transition.

The complete module, with the transitions themselves follows: first, the BNF part of the specification with the definition of processes; then a label declaration with a *read-write component* that models the global variable ‘`sh`’; finally, the first transition specifies that process 0 (‘`prc 0`’) will *increment* the global variable: at the start of the transition, the value of the global variable is indexed by ‘`sh`’, while at the end of the transition, the value, now updated, is indexed by ‘`sh'`’, as we mentioned before. Process 1, similarly, decrements the variables. We opted to limit the value of the shared variable between values 0 and 10, to prevent it from growing to arbitrary values. Even though the specification uses the (built-in) set ‘`Int`’, no inclusion of its defining module ‘`INT`’ was necessary: MSDF includes *automatically* all modules that define the sets that appear on transitions and label declarations. Another feature of MSDF is the fact that explicit variable declaration is *not* necessary. However, all variables in transitions must be named after their corresponding set, with an optional ‘`'`’ or number as postfix, such as ‘`Int'`’ or ‘`Int1`’.

```
msos THREAD-GAME is
 Proc . Proc ::= prc Int . Label = {sh : Int, sh' : Int, ...} .
    Int' := (Int + 1), Int < 10                Int' := (Int - 1), Int > 0
 -- -------------------------------------   -- ----------------------------------------
    (prc 0) : Proc -{sh = Int,                   (prc 1) : Proc -{sh = Int,
                    sh' = Int', -}-> prc 0 .                     sh' = Int', -}-> prc 1 .
sosm
```

In order to actually execute this "thread game," one needs transitions that deal with *both* processes running concurrently. This is done in a *modular* way with the introduction of a module '`THREAD-GAME/UNFAIR`', as follows. The set '`UnfairSoup`' is an associative-commutative juxtaposition function that gathers both processes; the application of the transition rule '`run`' selects, non-deterministically (and, possibly, unfairly), one process to execute. Note that the non-determinism is a byproduct of the associativity-commutativity of the process "soup" and therefore only one transition rule is necessary to specify '`run`'. The label notation '...' that appears in the transitions labels both in the premise and in the conclusion of '`run`' indicates that the semantic components in these transitions are the same. Due to the fact that the global variable is modelled by a read-write component, this means in practice that any change to '`sh`', made by the selected process, propagates to the conclusion. The module '`THREAD-GAME/UNFAIR`' includes the module '`THREAD-GAME`', containing the definition of the behavior of the processes described above, using the '`see` m' syntax, where m is any module name.

```
                                         prc Int -{...}-> prc Int
 [run] -- -----------------------------------------------------------------
         ((prc Int) UnfairSoup) : UnfairSoup -{...}-> (prc Int) UnfairSoup .
```

Let us now describe two possible analyses of the specification of the thread game. First it must be noted that, due to some requirements of the Maude system [9], user input in *Maude MSOS Tool* must be done between parentheses. After loading both modules into MMT, one may execute the '`rewrite` t' command that accepts an initial state t and attempts to rewrite until no more application of rewrite rules are possible. Optionally, the command may accept an upper limit n on the number of rewrites with syntax '`rewrite [`n`]` t'.

In order to rewrite a configuration, both syntax and label must be combined using the operator '`<_,_>`'. The example below shows a possible example. Recall that MSDF configurations are *typed syntactic trees*, which are specified using the operator '`_:::_`'. The Maude engine, in this example, chose to always apply the rule for process 0, increasing the value of the shared variable, from five to eight, in three successive applications of that rule.[1]

```
(rewrite [3] < (prc 0 prc 1) ::: 'UnfairSoup, { sh = 5 } > .)
rewrites: 4407 in 34ms cpu (34ms real) (125932 rewrites/second)
rewrite in THREAD-GAME/UNFAIR : <(prc 0 prc 1)::: 'UnfairSoup,{sh = 5}>
result Conf : <(prc 0 prc 1)::: 'UnfairSoup,{sh = 8}>
```

Another possibility, specially in the case of a non-deterministic specification, is to *search* for all the reachable configurations, starting from an initial configuration t. This is done using the command '`search` t r p', where t is the initial configuration, a tuple built with operator '`<_,_>`' as above; r a *rewrite relation*, that could be either '`=>*`' to search for all configurations that are reachable with zero or more rewrites, '`=>+`' to search for all configurations that are reachable with one or more rewrites, and '`=>!`' to search for all configurations that are *terminal*, that is, no rewrite rule applies;

[1]The outputs shown throughout the paper are edited and sometimes even omitted when trivial due to space constraints. The URL `http://www.ic.uff.br/~frosario/sbmf05.msdf` contains all the specifications presented in this paper, the analysis and their unedited outputs.

and p is a pattern to be matched against the reachable configurations. In the following example a search for all reachable configurations with one or more rewrite steps is done. By using the pattern '`C:Conf`', we expect to match against any possible configuration using the relation '`=>+`'.

```
(search < (prc 0 prc 1) ::: 'UnfairSoup, { sh = 5 } > =>+ C:Conf .)
rewrites: 879 in 30ms cpu (20ms real) (29300 rewrites/second)
search in THREAD-GAME/UNFAIR : <(prc 0 prc 1)::: 'UnfairSoup,{sh = 5}> =>+ C:Conf .
Solution 1  C:Conf --> <(prc 0 prc 1) ::: 'UnfairSoup,{sh = 6}>
Solution 2  C:Conf --> <(prc 0 prc 1) ::: 'UnfairSoup,{sh = 4}> ...
Solution 10 C:Conf --> <(prc 0 prc 1) ::: 'UnfairSoup,{sh = 0}>
No more solutions.
```

Remaining syntactic features of MSDF specification not shown in this section include: from a set s defined in a specification, the sets $s\star$ and $s+$ are automatically available and correspond, respectively, to the set of *tuples* and *non-empty tuples* of elements from s; and parameterized sets such as finite lists, finite maps, and finite sets. Section 4 shows how the Maude model checker can also be applied to MSDF specifications and how abstraction techniques [6] can be used. Next, Section 3 discusses how *Maude MSOS Tool* was formally designed and implemented on top of the Maude system.

3 *Maude MSOS Tool* as a Formal Tool

A formal tool has a precise definition of its underlying concepts in terms of mathematical objects, as opposed to conventional tools implemented directly on a programming language in an *ad-hoc* way. One possible choice is to use a logic $\mathcal{L}$ to axiomatize such concepts [11]. Moreover, if $\mathcal{L}$ may be related to a *logical framework* $\mathcal{F}$, that is, a logic expressive enough to represent several different logics, existing efforts developed for $\mathcal{F}$ may be *reused* and applied to $\mathcal{L}$. Within the meta-theory of general logics [7], one among the the several logical frameworks in the literature [16, 34], these concepts become mathematically precise as follows: a logic is an object in a category of logics and a mapping ϕ between two logics $\mathcal{L}$ and $\mathcal{L}'$ is a morphism in such a category. Moreover, if ϕ is computable, thanks to a meta-result of Bergstra and Tucker [1], ϕ may be represented as Church-Rosser and terminating equations.

Rewriting logic is one such logical framework $\mathcal{F}$. Martí Oliet and Meseguer [21] were the first ones to describe how several logics and specification languages may be mapped to rewriting logic, and many others followed [22], thus characterizing rewriting logic both as a logical and semantic framework. A key feature in rewriting logic to allow the definition of formal tools is *reflection*, that is, reasoning about rewrite theories happens *within* rewriting logic itself. To make this concept precise, let us first define a theory T in rewriting logic as a triple $\langle \Sigma, E, R \rangle$, where Σ is the *signature* of a rewrite theory, E is the set of Church-Rosser and terminating equations, and R is the set of rules that are applied *modulo* the equations, that is, rewriting happens on the E *equivalence classes* of Σ-terms. Rewriting logic has a very simple *calculus* that allows it to mirror deduction in any finitary logic as rewriting inference. We may now define that rewriting logic is reflective in the precise sense that there is a finitely presented rewrite theory U such that for any finitely presented rewrite theory T, including U itself, one has the following equivalence: $T \vdash t \longrightarrow t' \Leftrightarrow U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle$, where $\overline{T}$ and $\overline{t}$ are terms representing T and t as data elements of U of respective types Theory and Term. Since U is representable in itself, one can achieve a "reflective tower" with an arbitrary number of levels of reflection since we have $T \vdash t \longrightarrow$

$t \Leftrightarrow U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle \Leftrightarrow U \vdash \langle \overline{U}, \overline{\langle \overline{T}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{U}, \overline{\langle \overline{T}, \overline{t'} \rangle} \rangle \ldots$ Thus, when rewriting logic is chosen as our target logical framework $\mathcal{F}$, the mapping ϕ becomes the (equationaly axiomatized) metafunction $\overline{\phi} : \mathit{Theory}_L \to \mathit{Theory}$ where Theory_L is the type for theories in the logic L and $Theory$ is the type for rewriting logic theories.

Rewriting logic is realized in several performance tools [3, 9, 14]. One of particular interest is the Maude system due to its built-in support to reflection, through metaprogramming. Maude's '`META-LEVEL`' module declarations include concrete sorts for *Theory* and *Term* and the so called *descent functions*, its metaprogramming interface. We summarize below the key functionality provided by '`META-LEVEL`': i) Maude terms are reified as elements of a data type '`Term`' of terms; ii) Maude theories are reified as terms in a data type '`Module`' of modules; iii) The process of reducing a term to normal form is reifed by a function '`metaReduce`'; iv) The process of applying a rule of a module to a subject term is reified by a function '`metaApply`'; v) The process of rewriting a term in a system module using Maude's default strategy is reified by a function '`metaRewrite`'; vi) Parsing and pretty-printing of a term in a module are also reified by corresponding metalevel functions '`metaParse`' and '`metaPrettyPrint`'.

Full Maude is a Maude specification that endows the (Core) Maude system with an extensible module algebra [12]. In Full Maude one may declare parameterized modules and theories,[2] support for object-oriented modules and an extensible infra-structure that allows one to build formal tools, extending (Core) Maude capabilities. This support includes, besides the extended module algebra, parsing facilities, persistence, command line, and pretty-printing. In summary, in order to extend Full Maude to create a formal tool for a language L, one needs to: i) Define a mapping $\overline{\phi}$: $\mathtt{Module}_{\mathrm{L}} \to$ `Module`, where $\mathtt{Module}_{\mathrm{L}}$ is a datatype that represents language L and `Module` is the datatype for Maude modules; ii) Define a parsing function that given a sequence of identifiers produces a term of type $\mathtt{Module}_{\mathrm{L}}$; iii) May extend Full Maude's (persistent) database to hold attributes specific to $\mathtt{Module}_{\mathrm{L}}$; iv) Define a pretty-printing function that given a term in `Module` displays a sequence of identifiers representing a term in $\mathtt{Module}_{\mathrm{L}}$; v) Define commands appropriate to the L domain.

The implementation of *Maude MSOS Tool* in Maude [5] followed precisely the sequence outlined in the previous paragraph. The mapping $\overline{\phi}$ is a semantics preserving mapping [25] from MSOS to rewriting logic, in the precise sense of a bisimulation between the models of the MSOS specification and the resulting rewrite theory, with $\overline{\phi}$ defined from terms in a datatype `MSOSModule`, for MSOS specifications, to terms in the `Module` datatype, for rewrite theories in rewriting logic. It has essentially components to handle: implicit and explicit module inclusion, BNF declarations, parameterized types, implicit variable declaration, and labeled transition rules.

BNF declarations essentially equates sets and functions in MSDF to sorts and operators in membership equational logic [4], which is a generalization of order sorted logic: subset inclusion becomes subsort inclusion and parameterized sets are converted into parameterized sorts. To handle implicit module inclusions, the compilation process searches for all modules that declare the sets that are used in label declarations and transitions and explicitly include them in the generated Maude modules. Variables are automatically created by extracting the set name from the variables, since set names are only allowed to consist of letters.

The handling of labeled transition rules produces, essentially, a special form of a rewrite theory from a MSOS specification where the transition rules from the MSOS specification are represented

[2]The meaning of a theory here is a module with loose semantics.

as conditional rewrite rules in the resulting rewrite theory with the labels of the transition rules represented as extensible *records* in the rewrite theory. The heart of the representation of a label as a record is to transform it into a *preorder* by considering the pre and $post$ label projections. Such projections are defined by cases on each possible label index: i) For a read-only index i both pre and $post$ projections produce the value bound to i; ii) For a read-write index i the pre projection returns the first projection of the pair bound to i and the $post$ projection produces the second projection of the pair bound to i; iii) For a write-only index i in a labeled transition t, the pre projection is the prefix of the monoid bound to i and the $post$ projection is the appending operation of the monoid applied to $pre(i)$ with the value bound to i in t. For a write-only index in a labeled transition in a premise, the pre projection is the identity value of the monoid and the $post$ projection remains as before.

In order to compile MSDF specifications *Maude MSOS Tool* must first parse the user input that arrives via Maude's '`LOOP-MODE`' mechanism as a stream of quoted-identifiers. This is done using the metafunction '`metaParse`' and extending Full Maude's handling of user input in the module '`DATABASE-HANDLING`'. This module is basically a dispatcher that finds the appropriate handler based on the type of user input. Specifically for MMT, it was necessary to extend this module to add a dispatcher to terms of type '`MSOSUserInput`'. Once the compilation begins, this term of sort '`MSOSUserInput`' must be preprocessed before any compilation takes place, including a two-phase parsing process to handle the fact that a MSDF specification contains *user-definable* syntax on its signature component. After this processing is finished, we are left with a term of sort '`MSOSModule`', that is compiled into a Full Maude module and inserted into Full Maude database of metarepresented modules. Both representations are stored: the MSDF metamodule and the compiled module. This allows the tool to, for example, pretty-print MSDF modules by iterating through the structure of the '`MSOSModule`' term and outputting quoted-identifiers that will be printed back to the user via Maude's '`LOOP-MODE`'.

With MSDF syntax and basic *Maude MSOS Tool* functionality exemplified and the tool's formal foundation outlined, we may now move to the modular specification and analysis of distributed algorithms in MMT. This is accomplished in Section 4.

4 Case Studies

This Section describes the use of *Maude MSOS Tool* and MSDF in the specification and analysis of distributed algorithms, applying the support outlined in Section 3. As we mentioned before, all the specifications and analyses in this paper, and in particular in this Section, may be downloaded from `http://www.ic.uff.br/~frosario/sbmf05.msdf`.

We specify in MSDF different distributed algorithms from [20], namely the dining philosophers, with different scheduling policies, Lamport's bakery algorithm, and and verify them using Maude.[3] Besides state search, exemplified in Section 2, Maude's LTL model checker is also used [13].

We begin by describing our general model of process, starting with a simple notion of *processes* and *process identifiers* similar to the thread game of Section 1. A process contains an integer as

[3]A specification for leader election on asynchronous ring, omitted here for brevity, is also available at `http://www.ic.uff.br/~frosario/sbmf05.msdf` which exemplifies a message passing communication model and more sophisticated analysis with the model checker.

its process identifier (pid) and an abstract data type that represents its local state (‘`St`’). The local state is dependent on the algorithm being specified, and will be mostly used on our specifications to record the state of the computation of a process, but it can also store temporary values that are local to a specific process throughout the execution. The first specification is the dining philosophers algorithm.

4.1 Dining Philosophers

This solution to Dijkstra’s “Dining Philosophers” problem as described in [20] is based on breaking the symmetry when each philosopher acquires its fork: philosophers with even pids first attempt to acquire the fork at their left, while philosophers with odd pids first attempt to acquire the fork at their right.

By definition, the right fork of a philosopher i has number i, and the left fork has number $i+1 \bmod n$. When there is a competition to acquire a fork, the pids of the competing philosophers are inserted on a queue present in each fork. As each philosopher is done with the fork, it removes its pid from the queue.

The MSDF specification is as follows. First we need to map each fork id to a list of pids to implement the queue on each fork. The set ‘`Pids`’ defines that list of pids, declared with syntax ‘`Pids = (Int) List .`’, while ‘`Queue`’ defines the map from integers (fork ids) to ‘`Pids`’, declared with syntax ‘`Queue = (Int, Pids) Map .`’. Such queue can actually be seen as a *semantic component*, used to control the process configuration “soup,” and thus be placed inside the label and not in the process “soup,” declared with syntax ‘`Label = {q : Queue, q' : Queue, ...} .`’.

The specification is parameterized by a constant ‘`n`’, which will be instantiated, through an equation, to the number of philosophers on the table.

Each philosopher is a process that may be in one of the philosophers states, declared using BNF syntax ‘`St ::=`’: i) testing if the right fork or left may be picked, declared as ‘`stest-right | stest-left`’; ii) releasing the right or left fork, declared as ‘`sreset-right | sreset-left`’; iii) entering and leaving the critical region ‘`scrit | sexit`’; iv) attempting to acquire the forks, declared as ‘`stry`’; v) thinking state, declared as ‘`srem`’; vi) two intermediate states, not specified for brevity.

The forthcoming transitions formalize the meaning of the above philosopher’s states. Let us discuss only the most significant transitions for odd-numbered processes. The even-numbered transitions are symmetric to the ones shown here. Initially, all philosophers are hungry and will attempt to acquire their forks, that is, all processes are in the state ‘`stry`’. Odd-numbered processes, selected with the predicate ‘`odd(`i`)`’, attempt to acquire their right forks thus changing state ‘`stest-right`’.

```
                    odd (Int)
-- ---------------------------------------------
prc (Int, stry) : Proc --> prc (Int, stest-right) .
```

Another significant rule is the following: if the fork is unavailable, the process put its pid on the queue, and go back to test if its pid reached the beginning of the queue, as the rule below shows. Functions ‘`insert-back`’ and ‘`first`’ operate on parameterized lists.

```
odd (Int), Pids := lookup (Int, Queue),
Pids' := if (not Int in Pids) then insert-back (Int, Pids) else Pids fi,
Queue' := (Int |-> Pids') / Queue,
St := if first (Pids') == Int then stest-left else stest-right fi
-- ---------------------------------------------------------------------
prc (Int, stest-right) : Proc -{q = Queue, q' = Queue', -}-> prc (Int, St) .
```

The remainder of the specification, omitted for brevity, works as follows: after acquiring its right fork, the philosopher attempts to acquire its left fork using a similar transition. Upon acquiring both forks, a philosopher moves to its critical region and proceeds to put the forks down, first the right, then the left and finally enters its '`srem`' state, which represents a philosopher thinking. The process moves from '`srem`' directly to '`stry`', indicating that right after thinking he becomes hungry again.

The above code is contained in a module named '`DP`', whose intended purpose is to specify the behavior of each philosopher and fork specifically. In order to actually execute or verify the algorithm, we need to make explicit our process *scheduling policy*. This is achieved *modularly* with the creation of a module named '`DP/UNFAIR`' that specifies what we shall call an *unfair scheduling* policy.

We follow ideas present in [24, 2] and create a set '`UnfairSoup`' that represents an associative-commutative "soup" of processes. A single process is a trivial soup. The evolution of the soup is done by non-deterministically (and potentially) selecting a process out of the "floating processes," using matching modulo associativity and commutativity, evaluating this process, and putting it back into the soup. This behavior is specified by the transition labeled '`unfair`'.

```
UnfairSoup . UnfairSoup ::= Proc . UnfairSoup ::= UnfairSoup UnfairSoup [assoc comm] .
                               prc (Int, St) -{...}-> prc (Int, St')
 [unfair] -- -----------------------------------------------------------------------
         (prc (Int, St) UnfairSoup) : UnfairSoup -{...}-> (prc (Int, St')) UnfairSoup .
```

With the '`DP/UNFAIR`' module we add means to execute and verify the specifications in a completely *modular* way; no rules on the original specification needs to be changed. Let us begin our verification showing that the specification, with this scheduling policy, is free from deadlock. This verification is done in a modular way, by creating a module name '`DP/SEARCH/UNFAIR`' that includes the desired scheduling policy.

The identification of a deadlock state may be done using a '`search`' command with the '`=>!`' rewrite relation (see Section 2) since a final state is a state in which no rule applies, meaning that the entire pool of processes is "halted" and cannot continue to evolve. Such search produces no solution.

The auxiliary function '`initial-conf`' (not shown for brevity) creates an initial configuration with the desired number n of philosophers. We may further test the specification using more searches. For example we know that, in a configuration with four philosophers, two philosophers may eat at the same time, that is, to be at their respective '`scrit`' states. However, a philosopher may never eat concurrently with his neighbor. We may verify this property by querying our model with a '`search`' for all states in which two philosophers are in their '`scrit`' states:

```
search in DP/SEARCH : initial-conf =>*
 < (prc(I1:Int,scrit)prc(I2:Int,scrit) S:Soup)::: 'Soup, R:Record > .
I1:Int <- 0 ; I2:Int <- 2     I1:Int <- 1 ; I2:Int <- 3     I1:Int <- 2 ; I2:Int <- 0
```

The choice for the scheduling policy in module '`DP/UNFAIR`' does not have justice: nothing in the transition rule labeled '`unfair`' prevents from one or more philosophers to starve, or a single philosopher to eat continuously, due to the generality of the rewriting logic calculus. (See Section 5 for a note on support for strategies.)

We exemplify the use of Maude's built-in LTL model checking capabilities show that '`DP/UNFAIR`' scheduling is actually unfair as the module's name implies. For this purpose, another module was created, ‘DP/MODEL-CHECK’, that includes ‘DP/SEARCH’ and ‘MODEL-CHECKER’ modules. The former being a Maude module distributed with the Maude system that contains the signature for built-in model checking primitives. To use ‘MODEL-CHECKER’ functions, we need to specify which is our *state space* by declaring ‘Conf’, the sort for configurations, as a subsort to the built-in sort ‘State’.

Next, we create a proposition ‘state(i,s)’ that holds when process i is in state s. The model checking of the LTL formula ‘<> state (0,scrit)’, which means “eventually process 0 will be in state ‘scrit’”, fails. Looking at the counterexample, we notice that process 0 is “stuck” in state ‘sleave-try’ while process 2 keeps entering and leaving its critical region indefinitely.

```
rewrites: 3539 in 60ms cpu (60ms real) (58983 rewrites/second)
reduce in MODEL-CHECK : modelCheck(initial-conf,<> state(0,scrit))
result ModelCheckResult : counterexample(
  { prc(0,stry) prc(1,stry) prc(2,stry) prc(3,stry)}...,
  { prc(0,sleave-try) prc(2,srem) ... }       { prc(0,sleave-try) prc(2,stry) ... }
  { prc(0,sleave-try) prc(2,stest-left) ... }  { prc(0,sleave-try) prc(2,stest-right) ... }
  { prc(0,sleave-try) prc(2,sleave-try) ... }  { prc(0,sleave-try) prc(2,scrit) ... }
  { prc(0,sleave-try) prc(2,sexit) ... }       { prc(0,sleave-try) prc(2,sreset-left) ... }
  { prc(0,sleave-try) prc(2,sreset-right) ... }{ prc(0,sleave-try) prc(2,sleave-exit) ... }
```

Due to our modular design we may now *replace* our scheduling policy, keeping the dining philosophers specification intact. We now analyze the behavior of two different types of scheduling: round-robin and random.

The module ‘DP/FAIR/ROUND-ROBIN’ adds, as the name implies, a round-robin-type of scheduling: a global counter holds the identity of the next process to run, represented by a read-write label component named ‘fair’. The unprimed projection of label component ‘fair’ holds the process id of the process chosen to evolve and the primed projection of label component ‘fair’ holds the process id of the next process that may evolve. In transition rule labeled ‘f-rr’, given below, the value of the unprimed projection of component ‘fair’ is captured by the variable ‘Int’ and the primed projection by variable ‘Int′’, calculated as the successor of ‘Int’ modulo the number of processes in the “soup.” A process is chosen from the soup through associative-commutative matching of the process id of a process from the soup (‘prc (Int, St)’) with the value (‘Int’) bound to the unprimed projection of the ‘fair’ component.

```
Label = {fair : Int, fair' : Int, ...} .
FairSoup .  FairSoup ::= Proc .  FairSoup ::= FairSoup FairSoup [assoc comm] .
       Int' := (Int + 1) rem n,
                     prc (Int, St) -{fair = Int, fair' = Int,  ...}-> prc (Int, St')
[f-rr] -- -------------------------------------------------------------------------------------
(prc (Int, St) FairSoup) : FairSoup -{fair = Int, fair' = Int', ...}-> (prc (Int, St')) FairSoup .
```

Since each philosopher will execute in turns, it is not hard to believe that eventually a process will enter its critical region. The LTL formula ‘<> state(0, scrit)’ holds in our model.

It is important to ascertain whether the addition of our round-robin scheduling brought any problems. We may search again, if Maude manages to find a deadlocked state. Such search produces, again, no solution.

Round-robin scheduling may be over restrictive: each process will always execute in exactly the *same* order. We now turn to a second type of scheduling, where all philosophers are forced to eat at each “round”, but the order they eat varies from round to round. The following module, ‘DP/FAIR/RANDOM’, implements this policy by using an additional read-write component ‘runq’, the *run queue*, used to make sure that every process will have a chance in a round. The run queue contains the processes ids from each process from the process soup. The head of the run queue contains the id of the next process to run. When a process runs, it removes its id from the head of '`runq`' and insert it at the end of the queue. (This behavior is specified by transition rule labeled '`f-ra-1`'.) We also maintain a global counter, '`fair`', that registers how many processes have run. When this number reaches n, the number of processes, we begin a new round, *shuffling* the run queue with an externally defined function '`shuffle`'. (This behavior is specified by transition rule labeled '`f-ra-2`'.)

```
 Label = {fair : Int, fair' : Int, runq : Int*, runq' : Int*, ...} .
 RandomFairSoup . RandomFairSoup ::= Proc .
 RandomFairSoup ::= RandomFairSoup RandomFairSoup [assoc comm] .
 Int* ::= shuffle (Int*) .
             Int' := (Int + 1), Int < n,
             prc (Int1, St) -{fair = Int, fair' = Int,
                              runq  = (Int1, Int*), runq' = (Int1, Int*), ...}-> prc (Int1, St')
 [f-ra-1] -- -------------------------------------------------------------------------------------
             (prc (Int1, St) RandomFairSoup) : RandomFairSoup
               -{fair = Int, fair' = Int', runq  = (Int1, Int*), runq' = (Int*, Int1), ...}->
             (prc (Int1, St')) RandomFairSoup .

             Int == n
 [f-ra-2] -- --------------------------------------------------------------------------------------
             RandomFairSoup : RandomFairSoup
             -{fair = Int, fair' = 0, runq = Int*, runq' = (shuffle (Int*)), -}-> RandomFairSoup .
```

If we rerun both verifications described above, namely the search for a deadlock state and the LTL model checking for critical region reachability, again, no solution is found by the search ('`search initial-conf =>! C:Conf`') and the LTL formula for a process eventually reaching the critical section holds when model checked (`modelCheck(initial-conf,<> state(0, scrit))`).

Since the specification of the algorithm is separated from the verification module we may check our verification capabilities by providing an *erroneous specification*, with a deadlock, and checking if the tools provided by Maude are really able to find it. Also, we will have an opportunity to check if our scheduling policies have an influence on the discovery of the "bug."

To create a dining philosophers specification with deadlock is simply a matter of making the algorithm identical for odd- and even-numbered philosophers since there is no symmetric solution for the dining philosophers [20]. We choose a set of transitions rules, say the odd ones, as the only set of transitions in the specification and, of course, remove that condition from the premises of each transition. Replacing the "correct" '`DP`' module by this, nothing else needs to be changed and we may now rerun our tests. First, we begin by searching for a final state on each of the three possible scheduling policies. Notice that, indeed, in the three different policies, the problem is the same: deadlock when all philosophers acquire their left forks at the same time.

```
rewrites: 105900 in 1160ms cpu (1170ms real) (91293 rewrites/second)
search in DP/SEARCH/UNFAIR : initial-conf =>! C:Conf .
Solution 1 C:Conf -->
<(prc(0, stest-left) prc(1, stest-left) prc(2, stest-left) prc(3, stest-left)) ::: 'UnfairSoup,
{ q =(0 |->[0] +++ 1 |->[1]+++ 2 |->[2]+++ 3 |-> [3])}>
No more solutions.
rewrites: 12324 in 980ms cpu (980ms real) (12575 rewrites/second)
search in DP/SEARCH/FAIR/RANDOM : initial-conf =>! C:Conf .
Solution 1 C:Conf -->
<(prc(0, stest-left) prc(1, stest-left) prc(2, stest-left) ... prc(99, stest-left)) :::
'RandomFairSoup, { fair = 0,runq = 24,74 ... 99,49, q =(0 |-> [0] +++ ... +++ 99 |->[99])}>
No more solutions.
rewrites: 18703 in 6830ms cpu (6830ms real) (2738 rewrites/second)
search in DP/SEARCH/FAIR/ROUND-ROBIN : initial-conf =>! C:Conf .
Solution 1
```

```
C:Conf -->
<(prc(0, stest-left) prc(1, stest-left) prc(2, stest-left) prc(3, stest-left) ...
prc (399, stest-left)) ::: 'FairSoup, { fair = 0,q =(0 |->[0] +++ ... +++ 399 |->[399])}>
No more solutions.
```

4.2 Lamport's Bakery algorithm

This Section discusses a specification of Lamport's Bakery Algorithm, as described in [20]. With the primary objective of to exemplify the application of a technique named abstraction [6, 27] to an unbounded (infinite state) algorithm, in a modular way. Abstraction means to structure the state space, collapsing states together, and thus shortening the state space. This is accomplished in our specification by means of equations. The Bakery algorithm, intuitively, simulates a bakery where customers pick tickets when they enter and are served in the order given by their ticket numbers. Let us proceed to briefly describe the specification details of the algorithm.

When a process wants to enter its critical region (or be served by the bakery), it tells others that it is doing so by changing a global variable (`ch` in our specification). The process then chooses a number that is greater than all the numbers chosen by the other processes. This is done while the process is in its 'choosing(i, m)' state, where i is the number of processes left to check and m the greatest number found so far. While in the `choosing` state, a process must ignore its own number and keep the greatest number found so far, decrementing i as it passes through each other process. When $i = -1$, the process has exhausted its search and the greatest number found is m. The process then chooses $m + 1$ as its own number and goes to the next phase of the algorithm.

On this next phase, a process keeps a constant watch on the other processes, iterating through its 'waiting(i)' state, where n is the number of processes and $0 \leq i \leq n - 1$. It waits until its number is the lowest of all to enter its critical region and avoids comparison with any process that is currently choosing its own number.

Since there is a possibility that *several* processes begin the choice process at the same time, it may happen that processes choose the same number. The comparison, then, to find the lowest number, is made lexicographically using (i, p) where i is the process number and p its pid. Upon exiting its critical region, a process changes its chosen number to zero and moves to its remainder state. Once in its remainder state, a process may attempt to enter the critical region again.

Unfortunately, this algorithm does not have an upper bound on the chosen number. Moreover, the apparently trivial solution of using integers modulo some very large b also fails, as we shall demonstrate.

We may verify that there is no upper bound on the chosen number using a '`search`', showing that, with two processes, the chosen number can easily reach any natural number. The problem happens when a process chooses a number while the other process is in its critical region. A process only zeros its chosen number *after* leaving the critical region. It works as follows: process 0, with a chosen number of 2, is in its critical region; process 1 chooses 3 as its number; when this process is in its critical region, process 0 gets another number, which is 4, and so on. Below, we are looking for a state in which process 0 has chosen the number 10, while process 1 may have chosen any other number. This is represented by the record expression '`{PR:PreRecord, n = (0 |-> 10 +++ 1 |-> I:Int)}`', where '`PR`' is a variable that holds "the rest" of the record.

```
search [1] in BAKERY : initial-conf =>* < S:Soup,{PR:PreRecord,n = (0 |-> 10 +++ 1 |-> I:Int)} > .
... < (prc(0, choosing(0, 9)) prc(1, waiting(0))), {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, choosing(-1, 9)) prc(1, waiting(0))), {n = (0 |-> 0 +++ 1 |-> 9)} >
< (prc(0, waiting(0)) prc(1, waiting(0))), {n = (0 |-> 10 +++ 1 |-> 9)} >
```

In order to make this algorithm amenable to verification, we must create an abstraction that captures the essence of the algorithm, but does not have the infinite number of states of the original. The solution follows the ideas described in [27], in which a two-process abstraction is defined and proved to correctly simulate the original specification.

The key to find the correct abstraction in this case is to realize that the actual *absolute value* of the chosen number is not important, but its *relative value* with regard to the other numbers. A process changes its number to zero after leaving the critical zone, so the number chosen by the other process in this case does not need to grow indefinitely: choosing number one is sufficient. This is specified by two symmetrical equations, one for each process. One such equation is '`ceq (< S:Soup, n = (0 |-> 0 +++ 1 |-> I), PR >) = < S:Soup, n = (0 |-> 0 +++ 1 |-> 1), PR > if I > 1 .`', which specifies the upper bound for process one. Additional equations, ommitted here for brevity, are required to keep the chosen numbers of both processes from growing indefinitely, while keeping their relative values.

With these abstractions if we try a search for a race condition no solution will be found ('`search initial-conf =>* <(prc(0,crit)prc(1,crit))::::'Soup, PR:PreRecord>`'). Also, both processes eventually reach their critical region, according to the results of the two searches below:

```
rewrites: 3463 in 36ms cpu (36ms real) (93609 rewrites/second)
search in CHECK : initial-conf =>* <(prc(0,crit)prc(1,St:St))::: 'Soup, {PR:PreRecord}> .
Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0), n =(0 |-> 1 +++ 1 |-> 1); St:St <- waiting(0)
rewrites: 3376 in 26ms cpu (26ms real) (125055 rewrites/second)
search in CHECK : initial-conf =>* <(prc(1,crit)prc(0,St:St))::: 'Soup, {PR:PreRecord}> .
Solution 1
PR:PreRecord <- ch =(0 |-> 0 +++ 1 |-> 0), n =(0 |-> 2 +++ 1 |-> 1); St:St <- waiting(0)
```

5 Final Remarks

This paper presented a technique and a tool for the modular specification and analysis of distributed algorithms using the *Maude MSOS Tool*. We have shown how the use of MSOS has provided us with the necessary support for the creation of truly reusable modules in which new functionality were added monotonically. Initially, a distributed algorithm is specified in a independent manner: environment specific issues, such as scheduling may be specified at a later point, overcoming, for instance, the lack of fairness. *Maude MSOS Tool* allows for the execution by rewrite, state space search, and model checking of MSDF specifications within quite acceptable times as the experiments run so far [5] have shown, including those presented in this paper.

It is also important to assess the features described in this paper in light of those already available on the literature. Let us analyze, then, two significant examples of operational semantics tools: Hartel's LETOS [17], Pettersson's RML [35]. We begin by describing LETOS, "A Lightweight Execution Tool for Operational Semantics." The tool is written in C with a lex and yacc parser and uses a superset of Miranda [38] to specify operational and denotational semantics specifications, that are converted into Miranda scripts. An additional feature of LETOS is its support for pretty-printing specifications in LaTeX and to provide execution tracing using HTML pages. LETOS has partial support for non-deterministic specifications, simulated by functions returning lists, and the final result of a non-deterministic specification will be always only one of the possible final values. Abstract syntax is specified using Miranda's user-defined data type syntax. As is usual in operational semantics specifications [33], LETOS allows the definition of several different relations between configurations. No support of simulation (search) and model checking is present in the tool. Also, LETOS does not support MSOS (or any form of modularization whatsoever), hence, restricting its use for the purposes described in this paper.

Pettersson's Relational Meta-Language (RML) provides support for natural semantics [19] specifications. The RML system is a compiler written in Standard ML that compiles RML into efficient low-level C code. Like LETOS, RML supports creating different relations to be used on transitions. Unlike LETOS, RML does not have support for pretty-printing of specifications or tracing executions. Although RML supports splitting a specification into modules, it is a compiler specifically designed for natural semantics, and carries with it all the shortcomings of this formalism [32]. The use of a "big-step" style of specification precludes the specification of concurrent systems, something that is more naturally done using interleaving in "small-step" operational semantics styles. Having access to the small steps of execution of each process also allows the possibility of simulating and model checking specific details of concurrent systems. Finally, RML is a compiler for specifications, generating executable code; no support of simulation or model checking is present.

Logic programming can be done with Mosses's MSOS Tool Prolog implementation [31]. Even though modularity is supported, efficiency is quite poor and lacks support to model checking.

It is important to compare our approach to those present on tools made specifically for model checking concurrent systems. Let us briefly analyze two significant examples: SMV [23] and SPIN [18]. The main difference from our approach is the fact that the specification language is user-definable, whereas the two tools use a fixed specification language. This forces the use to move from its application-domain syntax (manually or otherwise) to the model-checker-domain language. SMV has support for splitting the specification into modules, but does not give any support for the modular specification style present on MSOS.

The main focus of future work is the integration of MMT with other formal tools available for the Maude environment; an initial experiment, still in prototype phase, was the integration with Alberto Verdejo's Strategy Language interpreter for Full Maude [26] making possible the use of *strategies* to control the rewriting process.

References

[1] J. A. Bergstra and J. V. Tucker. A characterization of computable data types by means of a finite, equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium, Noordwijkerhout*, volume 85 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, New York, NY, 1980.

[2] G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.

[3] 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.

[4] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[5] 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`.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[7] M. Clavel. *Reflection in General Logics and in Rewriting Logic, with Applications to the Maude Language*. PhD thesis, Universidad de Navarra, Spain, February 1998.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude as a metalanguage. In C. Kirchner and H. Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 237–250. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[9] M. Clavel, F. Durán, S. Eker, N. Martí-Oliet, P. Lincoln, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1)*. SRI International and University of Illinois at Urbana-Champaign, `http://maude.cs.uiuc.edu`, March 2004.

[10] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. M. Wing, J. Woodcock, and J. 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.

[11] 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.

[12] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Mlaga, Escuela Tcnica Superior de Ingeniera Informtica, 1999.

[13] 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.

[14] K. Futatsugi and R. Diaconescu. Cafeobj report. *World Scientific, AMAST Series*, 1998.

[15] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA, USA, 1996.

[16] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

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

[18] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[19] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[20] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[21] 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`.

[22] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002.

[23] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[24] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.

[25] 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. ISSN 0302-9743, ISBN 3-540-22381-9.

[26] J. Meseguer, N. Martí-Oliet, and A. Verdejo. Towards a strategy language for Maude. In N. 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.

[27] 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*. Springer-Verlag, 2003.

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

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

[30] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (Revised)*. MIT Press, 1997.

[31] 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.

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

[33] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing. John Wiley & Sons, Chichester, England, 1992.

[34] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

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

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

[37] J. Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[38] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architectures, Nancy, France*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, New York, NY, September 1985.