Maude: Specification and Programming in Rewriting Logic ⋆

M. Clavel [a], F. Durán [b], S. Eker [c], P. Lincoln [c], N. Martí-Oliet [d], J. Meseguer [c], and J. F. Quesada [e]

[a] *Departamento de Filosofía, Universidad de Navarra, Spain*

[b] *ETSI Informática, Universidad de Málaga, Spain*

[c] *SRI International, Menlo Park, California, USA*

[d] *Facultad de Matemáticas, Universidad Complutense, Madrid, Spain*

[e] *Centro de Informática Científica de Andalucía, Sevilla, Spain*

Abstract

Maude is a high-level language and a high-performance system supporting executable specification and declarative programming in rewriting logic. Since rewriting logic contains equational logic, Maude also supports equational specification and programming in its sublanguage of functional modules and theories. The underlying equational logic chosen for Maude is membership equational logic, that has sorts, subsorts, operator overloading, and partiality definable by membership and equality conditions. Rewriting logic is reflective, in the sense of being able to express its own metalevel at the object level. Reflection is systematically exploited in Maude endowing the language with powerful metaprogramming capabilities, including both user-definable module operations and declarative strategies to guide the deduction process. This paper explains and illustrates with examples the main concepts of Maude's language design, including its underlying logic, functional, system and object-oriented modules, as well as parameterized modules, theories, and views. We also explain how Maude supports reflection, metaprogramming and internal strategies. The paper outlines the principles underlying the Maude system implementation, including its semicompilation techniques. We conclude with some remarks about applications, work on a formal environment for Maude, and a mobile language extension of Maude.

Key words: Maude, rewriting logic, functional modules, system modules, parameterization, reflection, internal strategies.

⋆ Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Re-

1 Introduction

Maude [14,15] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude has been influenced in important ways by OBJ3 [37]; in particular, Maude's equational logic sublanguage essentially contains OBJ3 as a sublanguage. The main differences from OBJ3 at the equational level are a much greater performance, and a richer equational logic, namely, membership equational logic [48], that extends OBJ3's order-sorted equational logic [36].

The key novelty of Maude is that—besides efficiently supporting equational computation and algebraic specification in the OBJ style—it also supports *rewriting logic* computation. Rewriting logic [43] is a logic of concurrent change that can naturally deal with state and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency [47,35,11,53]. In particular, it supports very well concurrent *object-oriented* computation. This is reflected in Maude's design by providing special syntax for *object-oriented modules*. Since the computational and logical interpretations of rewriting logic are like two sides of the same coin, the same reasons making it a good semantic framework at the computational level make it also a good *logical framework* at the logical level, that is, a *metalogic* in which many other logics can be naturally represented and implemented [41]. Consequently, some of the most interesting applications of Maude are *metalanguage* applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation.

Maude's *functional modules* are theories in *membership equational logic* [48,9], a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : s$, stating that a term t has sort s. Such a logic extends order-sorted equational logic [36], and supports sorts, subsort relations, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains. Maude's functional modules are assumed to be Church-Rosser and terminating; they are executed by the Maude engine according to the rewriting techniques and operational semantics developed in [9].

Membership equational logic is a sublogic of *rewriting logic* [43]. A rewrite theory is a pair (T, R) with T a membership equational theory, and R a collection of labelled and possibly conditional *rewrite rules* involving terms in the signature of T. Maude's *system modules* are rewrite theories in exactly this

search Contract N00014-99-C-0198, and by National Science Foundation Grant CCR-9900334.

sense. The rewrite rules $r : t \longrightarrow t'$ in R are *not* equations. Computationally, they are interpreted as local *transition rules* in a possibly concurrent system. Logically, they are interpreted as *inference rules* in a logical system.

Rewriting in (T, R) happens *modulo* the equational axioms in T. Maude supports rewriting modulo all combinations of associativity, commutativity, and identity. The rules in R need not be Church-Rosser and need not be terminating. Many different rewriting paths are then possible; therefore, the choice of appropriate *strategies* is crucial for executing rewrite theories. In Maude, such strategies are not an extralogical part of the language. They are instead *internal strategies* defined by rewrite theories at the metalevel. This is because rewriting logic is *reflective* [12,20] in the precise sense of having a *finitely presented universal theory* U that can simulate any finitely presented rewrite theory. Since U is representable in itself, we can then achieve a "reflective tower" with an arbitrary number of levels of reflection.

Maude efficiently supports this reflective tower through its `META-LEVEL` module, which makes possible not only the declarative definition and execution of user-definable rewriting strategies, but also many other metaprogramming applications. In particular, it is possible to define and execute within the logic an extensible *module algebra* supporting the OBJ style of *parameterized programming* [37], with highly generic and reusable modules. The basic idea is that `META-LEVEL` is extended with new data types for: *parameterized* modules; *theories*, with loose semantics, to state formal requirements in parameters; *views*, to bind parameter theories to their instances; and *module expressions*, instantiating, transforming, and composing parameterized modules. All such new types and operations are defined in Maude itself. This, together with the explicit access to modules as terms provided by reflection, makes the corresponding module algebra completely open, and easily extensible by new module operations and transformations [28]. Maude also supports *object-oriented* modules, with convenient syntax for object-oriented applications.

All applications typical of equational programming and algebraic specification are conveniently and efficiently supported through Maude's sublanguage of functional modules. In fact, the paper [48] argues that Maude's equational logic, namely, membership equational logic, is so expressive—yet efficiently implementable—as to offer very good advantages as a logical framework for a very wide range of algebraic specification languages based on both total and partial equational logic formalisms. However, many Maude applications go beyond equational logic. System modules support general rewriting logic applications. The important area of concurrent and distributed object-based system specification and prototyping is supported by object-oriented modules. In addition, reflection makes possible many novel metaprogramming and metalanguage applications, and is extremely valuable in the use of rewriting logic as a *logical and semantic framework* [41].

The rewriting logic research program has shown good signs of vitality, including three international workshops [46,39,34], over two hundred research papers (see the references in [47,49,51,50]), and three language implementation efforts, namely ELAN [40,8,7] in France, CafeOBJ [23,25,24] in Japan, and Maude. Therefore, Maude should be seen as our contribution to the broader collective effort of building good language implementations for rewriting logic. In this regard, a key distinguishing feature of Maude is its systematic and efficient use of *reflection*, exploiting the fact that rewriting logic is reflective, a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications.

This paper constitutes a revised and extended presentation of concepts and ideas previously introduced in several conference papers [19,13,30,16,31]. Those papers have provided snapshots of the language versions at different moments, while this journal version focuses on the main concepts in a (mostly) version-independent way. However, we do not develop here complete presentations of the underlying logics, providing instead bibliographic references where the reader can find more details.

The reader is assumed to have some knowledge of algebraic specification concepts (as surveyed for example in the recent book [1]). For a more introductory presentation of Maude, the reader is advised to read the Maude tutorial [15], where the main features of the language are introduced in an incremental way by means of a sequence of detailed examples. More language details can also be found in the Maude manual [14], which has large amounts of version-dependent information. We plan to keep the manual as an evolving online document reflecting new versions of the language as they are developed.

The Maude system, the just mentioned tutorial and manual, a collection of examples and case studies, and a list of related papers are available (free of charge) at `http://maude.csl.sri.com`.

2 Membership Equational Logic and Functional Modules

Maude is a declarative language based on rewriting logic, but rewriting logic has its underlying equational logic as a parameter. There are, for example, unsorted, many-sorted, and order-sorted versions of rewriting logic, each containing the previous version as a special case. In particular, the underlying equational logic chosen for Maude is *membership equational logic*, a conservative extension of both order-sorted equational logic and partial equational logic with existence equations [48,9]. It supports partiality, subsort relations, operator overloading, and error specification.

2.1 Membership Equational Logic

A *signature* in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, (K, Σ) a many-sorted (although it is better to say "many-kinded") signature, and $S = \{S_k\}_{k \in K}$ a K-kinded set of *sorts*. An Ω-*algebra* is then a (K, Σ)-algebra A together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Intuitively, the elements in sorts are the good, or correct, or non-error, or defined, elements, whereas the elements without a sort are error or undefined elements. In general, a total function at the kind level restricts only to a *partial* function at the level of sorts.

Atomic formulas are either Σ-*equations*, or *memberships* of the form $t : s$, where the term t has kind k and $s \in S_k$. General sentences are Horn clauses on these atomic formulas, quantified by finite sets of K-kinded variables. That is, they are either *conditional equations*

$$(\forall X) \;\; t = t' \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or *conditional memberships* of the form

$$(\forall X) \;\; t : s \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Such memberships are a generalization of sort constraints [52] and can be used to specify partial functions, that become defined when their arguments satisfy certain equational and membership conditions.

Order-sorted notation can also be used for convenience, and we do so in Maude. Thus, a subsort declaration $s < s'$ abbreviates the conditional membership axiom $(\forall x) \;\; x : s' \;\; \mathit{if} \;\; x : s$. Similarly, an operator declaration $f : s_1 \ldots s_n \to s_0$ at the sort level corresponds to an operator declaration at the kind level together with the conditional membership axiom $(\forall x_1, \ldots, x_n) \;\; f(x_1, \ldots, x_n) : s_0 \;\; \mathit{if} \;\; x_1 : s_1 \wedge \ldots \wedge x_n : s_n$.

Membership equational logic has all the usual good properties: soundness and completeness of appropriate rules of deduction, initial and free algebras, relatively free algebras along theory morphisms, and so on [48].

2.2 Functional Modules

In Maude, *functional modules* are equational theories in membership equational logic satisfying some additional requirements. Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found. This is the reason why the equations must satisfy the additional requirements of being Church-Rosser, terminating, and sort decreasing [9]. This guarantees that all terms in an equivalence class modulo the equations will rewrite to a unique canonical form, and that this canonical form can be assigned a sort that is smaller than all other sorts assignable to terms in the class. Since Maude supports rewriting modulo equational theories such as associativity, commutativity, and identity, all that we say has to be understood for equational rewriting *modulo such axioms* [22].

We explain now the syntactic treatment in Maude of kinds, variables, and conditions in conditional equations and membership axioms.

With respect to kinds, Maude does automatic kind inference from the sorts declared by the user and their subsort relations, but kinds are not explicitly named; instead, a kind k is identified with the set S_k of its sorts, interpreted as an *equivalence class* modulo the equivalence relation generated by the subsort ordering, that is, two sorts are in this equivalence relation if and only if they belong to the same connected component in the poset of sorts. Therefore, for any $s \in S_k$, $[s]$ denotes the kind $k = S_k$, understood as the connected component of the poset of sorts to which s belongs.

As an example that will be developed step by step in this section, let us consider as given a graph specification

```
sorts Node Edge .
ops source target : Edge -> Node .
```

with operations giving the source and target nodes of each edge, as well as specific edge and node constants that need not concern us here. Then, we extend such a specification by declaring a sort `Path` of paths over the graph, together with a *partial* concatenation operator, and appropriate source and target functions over paths as follows, where the subsort declaration states that edges are "unitary" paths.

```
sort Path .
subsort Edge < Path .
op _;_ : [Path] [Path] -> [Path] .
ops source target : Path -> Node .
```

This illustrates the idea that in Maude sorts are user-defined, while kinds are implicitly associated with connected components of sorts and are considered as "error supersorts." The Maude system also lifts automatically to kinds all the operators involving sorts of the corresponding connected components to form *error expressions*. Such error expressions allow us to give expressions to be evaluated the benefit of the doubt: if, when they are simplified, they have a legal sort, then they are ok; otherwise, the fully simplified error expression is returned as an error message.

Variables in a Maude module do not have to be declared in variable declarations; they can appear directly in terms. A variable consists of an identifier composed of a name, followed by a colon, followed by either a sort or a kind name. For example, `P:Path` is a variable of sort `Path`. Variable declarations are still allowed for convenience; for example, the declaration `var P : Path` allows using the name `P` as an abbreviation for the variable `P:Path`.

Equational conditions in conditional equations and memberships are made up of individual equations $t = t'$ and memberships $t : s$ by a binary conjunction connective `/\` which is assumed associative. Furthermore, the concrete syntax of equations in conditions has two variants, namely, ordinary equations `t = t'`, and *matching equations* `t := t'`.

For example, the following axioms express the condition defining path concatenation and the associativity of this operator:

```
var E : Edge .
vars P Q R S : Path .
cmb E ; P : Path if target(E) = source(P) .
ceq (P ; Q) ; R = P ; (Q ; R)
     if target(P) = source(Q) /\ target(Q) = source(R) .
```

The conditional membership axiom (introduced by the keyword `cmb`) states that an edge concatenated with a path is also a path when the target node of the edge coincides with the source node of the path. This has the effect of defining path concatenation as a partial function on paths, although it is total on the kind `[Path]` of “confused paths.” Instead of giving the above associativity equation explicitly (by means of the conditional equation introduced by the keyword `ceq`), if we wanted to apply the axioms *modulo associativity*, we could have declared an associativity equational attribute in the declaration of the operator:

```
op _;_ : [Path] [Path] -> [Path] [assoc] .
```

Assuming variables `P`, `E`, and `S` declared as above, source and target functions over paths are defined by means of matching equations in conditions as follows:

```
ceq source(P) = source(E) if E ; S := P .
ceq target(P) = target(S) if E ; S := P .
```

Matching equations are mathematically interpreted as ordinary equations; however, operationally they are treated in a special way and they must satisfy special requirements. Note that the variables `E` and `S` in the above matching equation do not appear in the lefthand sides of the corresponding conditional equations. In the execution of these equations, these new variables become instantiated by *matching* the term `E ; S` against the subject term bound to the variable `P`. In order for this match to decide the equality with the ground term bound to `P`, the term `E ; S` must be a *pattern*. Given a functional module M, we call a term t an M*-pattern* if for any well-formed substitution σ such that for each variable x in its domain the term $\sigma(x)$ is in canonical form with respect to the equations in M, then $\sigma(t)$ is also in canonical form. A sufficient condition for t to be an M*-pattern* is the absence of unifiers between its nonvariable subterms and lefthand sides of equations in M.

Ordinary equations $t = t'$ in conditions have instead the usual operational interpretation, that is, for the given substitution σ, $\sigma(t)$ and $\sigma(t')$ are both reduced to canonical form and compared for equality, modulo the equational axioms specified in the module's operator declarations such as associativity, commutativity, and identity.

All conditional equations $t = t' \;\; if \;\; C_1 \wedge \ldots \wedge C_n$ in a functional module M have to satisfy the following *admissibility requirements*,[1] ensuring that all the extra variables will become instantiated by matching:

(1) $$vars(t') \subseteq vars(t) \cup \bigcup_{j=1}^{n} vars(C_j).$$

(2) If C_i is an equation $u_i = u_i'$ or a membership $u_i : s$, then

$$vars(C_i) \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j).$$

(3) If C_i is a matching equation $u_i := u_i'$, then u_i is an M-pattern and

$$vars(u_i') \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j).$$

The satisfaction of the conditions is attempted sequentially from left to right. Since matching takes place modulo equational attributes, in general many different matches may have to be tried until a match of all the variables satisfying the condition is found.

As mentioned before, we expect functional modules to be Church-Rosser and terminating membership equational logic specifications in the sense of [9, Section 10.1]. The above admissibility requirements and the Church-Rosser and termination assumptions are dropped for functional *theories* (see Section 4.2) which support the full generality of the logic.

[1] These requirements include as a special case what are called *properly oriented and right stable 3-CTRSs* in [61], when each equation $s_i = t_i$ in their conditions is expressed as a matching equation $t_i := s_i$.

In membership equational logic the Church-Rosser property of terminating and sort-decreasing equations is indeed equivalent to the confluence of their critical pairs in an appropriately generalized sense [9]. Furthermore, both equality and membership of a term in a sort are then *decidable* properties [9]. That is, the equality and membership predicates are *computable functions*. We can then use the metatheorem of Bergstra and Tucker [3] to conclude that such predicates are themselves specifiable by Church-Rosser and terminating equations as Boolean-valued functions. This has the pleasant consequence of allowing us to include inequalities $t \neq t'$ and negations of membership assertions $not(t : s)$ in conditions of equations and of membership axioms, since such seemingly negative predicates can also be axiomatized *inside the logic* in a positive way, provided that we have a subspecification of (not necessarily free) constructors in which to do it, and that the specification is indeed Church-Rosser, terminating, and sort decreasing. Of course, in practice they do *not* have to be explicitly axiomatized, since they are built into the implementation of rewriting deduction in a much more efficient way.

Indeed, by default, Maude modules implicitly import a predefined `BOOL` module providing Boolean values `true` and `false`, and operators `_and_`, `_or_`, and `not_`. In addition, this imported predefined module provides the semantic equality operator `_==_` checked by equational simplification, its negation `_=/=_`, a conditional operator `if_then_else_fi`, and a membership predicate `_:_`. For example, the associativity property could also be specified as

```
ceq (P ; Q) ; R = P ; (Q ; R)
   if target(P) == source(Q) and target(Q) == source(R) .
```

More generally, a Boolean expression `b` is allowed to appear as a conjunct in an equational condition as a shorthand for the equation `b = true`.

If a collection of (conditional) equations is Church-Rosser and terminating, given an expression, no matter how the equations are used from left to right as simplification rules, any reduction strategy will reach a normal form and moreover we will always reach the same final result. However, even though the final result may be the same, some orders of evaluation can be considerably more efficient than others. It may therefore be useful to have some way of controlling the way in which equations are applied by means of strategies.

Typically, a functional language is either eager, or lazy with some strictness analysis added for efficiency, and the user has to live with whatever the language provides. Maude adopts OBJ3's flexible method of user-specified evaluation strategies on an operator-by-operator basis [37], adding some improvements to the OBJ3 approach to ensure a correct implementation [33]. For an n-ary operator f such strategies are specified as lists $i_1 \ldots i_m$ of numbers, with $i_m = 0$, and $0 \leq i_j \leq n$, for $j = 1, \ldots, m-1$. For example, the default bottom-up eager strategy given in Maude to an n-ary operator f, when no strategy is explicitly declared by the user, is $(1 \ldots n\ 0)$, stating that in evaluating a term $f(t_1, \ldots, t_n)$, the subterms $t_1, \ldots, t_n$ are evaluated in this order before applying the equations for f to the whole term. Similarly, the strategy given to `if_then_else_fi` is (1 0 2 3 0), stating that it is enough to evaluate the Boolean condition in the first argument before trying the evaluation of the whole term. In addition to improving efficiency, operator strategies allow us to compute with infinite data structures which are evaluated on demand; for example, a lazy "cons" list constructor may have strategy (0). The paper [33] documents in detail the operational semantics and the implementation techniques for Maude's operator evaluation strategies; their concrete syntax as attributes in operator declarations is explained in [14].

As in the OBJ family of languages [37], functional modules can be unparameterized, or they can be parameterized with *functional theories* as their parameters (see Section 4 for more details). Functional theories are also membership equational logic theories, but they do not need to be Church-Rosser and terminating. They have a *loose* interpretation, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model. On the other hand, the semantics of an unparameterized functional module is the *initial algebra* specified by its theory. The semantics of a *parameterized* functional module is the free functor associated to the inclusion of the parameter theory into the body of the parameterized module [48,27]. For example, the semantics of a list module `LIST(X :: TRIV)` parameterized over the simple parameter theory `TRIV` with only one sort `Elt` (see Section 4.3) is the functor sending each set to the algebra of lists over this set. Similarly, the semantics of a sorting module `SORTING(Y :: POSET)` parameterized over the `POSET` functional theory (see Section 4.2) is the functor sending each poset to the algebra of lists for that poset with a sorting function.

2.3 Example: Arrays as Lists of Pairs

We finish this section with a functional module illustrating Maude's support for mixfix user-definable syntax and for module hierarchies (see Section 4.1).

An array of integers is represented as a list of pairs of integers, where the first component of each pair corresponds to the array position and the second to the value in that position. A list of pairs of this kind is the representation of an array if either it is empty, or the first components of the pairs are all different and the positions of consecutive pairs are consecutive numbers.

The first module imports the predefined module `MACHINE-INT`, providing integers and usual arithmetic operations on them. Then, it defines a sort `IntPair` for pairs of integers with (_,_) as only constructor[2] (notice the ctor attribute specifying that this operator is a constructor of the sort). These pairs are used as components of lists, defined with the concatenation operator __ as the main constructor, declared with both an attribute assoc for associativity, and an attribute id: nil for the empty list nil as two-sided identity. Unitary lists are obtained with a subsort declaration.

```
fmod INT-PAIR-LIST is
    protecting MACHINE-INT .
    sort IntPair .
    op ((_,_)) : MachineInt MachineInt -> IntPair [ctor] .
    sort IntPairList .
    subsort IntPair < IntPairList .
    op nil : -> IntPairList [ctor] .
    op __ : IntPairList IntPairList -> IntPairList
         [ctor assoc id: nil] .
endfm
```

The following module INT-ARRAY imports the previous one, and then uses (conditional) memberships to specify the subsort IntArray of lists representing arrays. The sort NeIntArray is the subsort of nonempty such lists. The module INT-ARRAY defines two usual partial operators on arrays: _[_] to obtain the value stored in the array at a given position, and _[_->_] to modify the value at a particular position. Notice that the partiality of such operators is reflected in their declarations as returning values in a kind instead of a sort. Finally, the operators low and high return, respectively, the first and last positions of a given nonempty array.

```
fmod INT-ARRAY is
    protecting INT-PAIR-LIST .
    sorts NeIntArray IntArray .
    subsorts IntPair < NeIntArray < IntArray < IntPairList .

    op _[_] : NeIntArray MachineInt -> [MachineInt] .
    op _[_->_] : NeIntArray MachineInt MachineInt -> [IntArray] .
    ops low high : NeIntArray -> MachineInt .

    vars I J X Y : MachineInt .
    vars L L' : IntPairList .

    mb nil : IntArray .
    cmb (I, X) (J, Y) L : NeIntArray
        if I + 1 = J  /\  (J, Y) L : NeIntArray .
```

[2] Since parentheses are normally used for disambiguation, in order to correctly declare this operation, it is necessary to write ((_,_)).

```
    ceq (L (I, X) L')[I] = X if L (I, X) L' : NeIntArray .
    ceq (L (I, X) L')[I -> Y] = (L (I, Y) L')
        if L (I, X) L' : NeIntArray .

    ceq low((I, X) L) = I if (I, X) L : NeIntArray .
    ceq high(L (I, X)) = I if L (I, X) : NeIntArray .
endfm
```

We remark that in all the conditional equations above there are memberships in the conditions, making sure that the arguments belong to the appropriate sorts. These checks guarantee that the equations are applied only to terms having a sort (in addition to having a kind, which is checked at parsing time) and therefore that computation takes place over "good" terms, since terms that fail to have a sort are considered "error" terms. Equations intended for error and exception recovery should not include such memberships in conditions.

3 Rewriting Logic and System Modules

The type of rewriting typical of functional modules terminates with a single value as its outcome. In such modules, each step of rewriting is a step of *replacement of equals by equals*, until we find the equivalent, fully evaluated value. In general, however, a set of rewrite rules need not be terminating, and need not be Church-Rosser. That is, not only can we have infinite chains of rewriting, but we may also have highly divergent rewriting paths, that could never cross each other by further rewriting.

The essential idea of rewriting logic [43] is that the *semantics* of rewriting can be drastically changed in a very fruitful way. We no longer interpret a term t as a functional expression, but as a *state* of a system; and we no longer interpret a rewrite rule $t \longrightarrow t'$ as an equality, but as a *local state transition*, stating that if a portion of a system's state exhibits the pattern described by t, then that portion of the system can change to the corresponding instance of t'. Furthermore, such a local state change can take place independently from, and therefore concurrently with, any other non-overlapping local state changes. Rewriting logic is therefore a logic of concurrent state change.

3.1 Rewriting Logic

A *signature* in rewriting logic is an equational theory (Ω, E), where Ω is an equational signature and E is a set of Ω-equations. Rewriting will operate on equivalence classes of terms modulo E: for example, string rewriting is obtained by imposing an associativity axiom; multiset rewriting by imposing associativity and commutativity; and standard term rewriting is obtained as the particular case in which the set of equations E is empty. Techniques for rewriting modulo equations have been studied extensively [22,38,55] and can be used to implement rewriting modulo many equational theories of interest. This is precisely what Maude does, using the equational attributes given in operator declarations—such as associativity, commutativity, and identity—to rewrite modulo such axioms.

Sentences over a signature (Ω, E) have the form $[t]_E \longrightarrow [t']_E$, where t and t' are Ω-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E (usually, we omit the subscript and simply write $[t]$). A *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Omega, E, L, R)$ where Ω is an equational signature, E is a set of Ω-equations, L is a set of *labels*, and R is a set of labelled *rewrite rules* either of the unconditional form $r : [t] \longrightarrow [t']$, or of the conditional form explained in Section 3.2.

Rewriting logic is a logic for reasoning about *concurrent systems* having *states*, and evolving by means of *transitions*. The signature of a rewrite theory describes a particular structure for the states of a system, and the rewrite rules describe which *elementary local transitions* are possible in the distributed state. The inference rules of rewriting logic [43] allow to deduce general concurrent transitions which are possible in a system satisfying such a description.

3.2 System Modules

The most general Maude modules are *system modules*. They specify the initial model $\mathcal{T}_\mathcal{R}$ of a rewrite theory $\mathcal{R} = (\Omega, E, L, R)$ in the membership equational logic variant of rewriting logic (for a detailed construction of $\mathcal{T}_\mathcal{R}$ in the unsorted case see [43]). These initial models capture nicely the intuitive idea of "rewrite systems" in the sense that they are transition systems whose states are equivalence classes $[t]$ of ground terms modulo the equations E in $\mathcal{R}$, and whose transitions are proofs $\alpha : [t] \longrightarrow [t']$ in rewriting logic, that is, concurrent rewriting computations in the system described by the rules in R. Such proofs are equated modulo a natural notion of proof equivalence that computationally corresponds to the "true concurrency" of the computations. By adopting a logical instead of a computational perspective, we can alternatively view such models as "logical systems" in which formulas are validly rewritten to other formulas by concurrent rewritings which correspond to proofs for the logic in question. These models have a natural *category* structure, with states (or formulas) as objects, transitions (or proofs) as morphisms, and sequential composition as morphism composition, and in them dynamic behavior exactly corresponds to deduction. In the parameterized case (see Section 4), the inclusion from the parameter(s) into the module then gives rise to a free extension functor [42], which provides the semantics for the module.

As a first example of system module, we consider an extension of the module defining integer arrays in Section 2.3.

```
mod INT-SORTING is
    protecting INT-ARRAY .
    vars I J X Y : MachineInt .
    var L : IntPairList .
    crl [sort] : (I, X) L (J, Y) => (I, Y) L (J, X)
       if X > Y  /\ (I, X) L (J, Y) : NeIntArray .
endm
```

For this system module, the corresponding rewrite theory (Ω, E, L, R) consists of: a signature Ω given by the sorts, subsort relations, and operator declarations in `INT-ARRAY`, along with a set of equations and memberships E also declared in `INT-ARRAY`; a label set L that only contains the label `sort`; and a set of rules R that consists of the conditional rule (introduced by the keyword `crl`) on integer arrays that exchanges two values when they are out of place. The system thus described is highly concurrent, since the `sort` rule can be applied concurrently to many different positions in the array. This specification happens to be confluent and terminating, but in general these properties do not hold for other system modules.

Computations need not be confluent (indeed, they can be highly nondeterministic) and need not be terminating. Therefore, the issue of *executing* rewriting logic specifications of system modules in general is considerably more subtle than executing expressions in a functional module, for which the termination and Church-Rosser properties guarantee a unique final result regardless of the order in which equations are applied as simplification rules. Hence, we need to have good ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate *strategies*. As we explain in Section 6, using reflection the rewriting inference process can be controlled with great flexibility in Maude by means of strategies that are defined by rewrite rules at the metalevel. However, the Maude interpreter provides a *default strategy* for executing expressions in system modules (see the end of this subsection).

At the equational level, system modules satisfy the same equational requirements already described for functional modules, including the requirement that the equations are Church-Rosser and terminating modulo the given equational axioms. Furthermore, rewrite rules can take the most general possible form in the variant of rewriting logic built on top of membership equational logic, that is, they are of the form

$$t \to t' \ \ if \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \to q_k)$$

with no restriction on which new variables may appear in the righthand side or the condition. That is, conditions in rules are also formed by an associative conjunction connective `/\`, but they generalize conditions in equations and memberships by allowing also rewrite expressions, for which the concrete syntax `t => t'` is used. Furthermore, equations, memberships, and rewrites can be intermixed in any order, and, as for functional modules, some of the equations in conditions can be matching equations.

Of course, in that full generality the execution of a system module will require *strategies* that control at the metalevel the instantiation of the extra variables in the condition and in the righthand side [12,66]. However, a quite general class of system modules, called *admissible modules*, are executable by Maude's default interpreter. As already mentioned, the equational part of a system module must always satisfy the same requirements given in Section 2.2 for functional modules; furthermore, as explained later in this section, its rules must be *coherent* with respect to its equations. A system module M is called *admissible* if, in addition to the above requirements, each of its rewrite rules

$$t \to t' \ \ if \ \ C_1 \wedge \ldots \wedge C_n$$

satisfies the admissibility requirements (1)-(3) in Section 2.2 plus the additional requirement

(4) If C_i is a rewrite $u_i \to u'_i$, then

$$vars(u_i) \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(C_j),$$

and u'_i is an $\mathcal{E}(M)$-pattern, for $\mathcal{E}(M)$ the equational theory underlying the module M.

Operationally, we try to satisfy such a rewrite condition by reducing the instance $\sigma(u_i)$ to canonical form v_i with respect to the equations, and then trying to find a rewrite proof $v_i \to w_i$ with w_i in canonical form with respect to the equations and such that w_i is a substitution instance of u'_i.

As for functional modules, when executing a conditional rule in an admissible system module, the satisfaction of all its conditions is attempted sequentially from left to right; but notice that now, besides the fact that many matches for the equational conditions may be possible due to the presence of equational axioms, we also have to deal with the fact that solving rewrite conditions requires *search*, including searching for new solutions when previous ones fail to satisfy subsequent conditions. The default interpreter supports search computations, in which the search is controlled by means of several parameters. In general, the conditions solved by the default interpreter may be conjunctions of rewrites, memberships, and equations, with appropriate restrictions on the occurrence of new variables in the conjuncts.

We illustrate Maude's syntax for system modules by means of an admissible module from [41] that defines the transition system semantics for Milner's CCS [54] in such a way that transitions correspond to rewrites; that is, a rewrite `P => {A}Q` means that process `P` has performed action `A` becoming process `Q`, which is usually written as $\mathtt{P} \xrightarrow{\mathtt{A}} \mathtt{Q}$. Full CCS is represented, including possibly recursive process definitions by means of contexts. The reader can find the modules defining the missing pieces of the syntax in Appendix A.1.

```
mod CCS-SEMANTICS-TRANS is
    protecting CCS-CONTEXT .
    sort ActProcess .
    subsort Process < ActProcess .
    op {_}_ : Act ActProcess -> ActProcess [ctor] .
    vars L M : Label .          var A : Act .
    vars P P' Q Q' : Process .  var X : ProcessId .

    *** Prefix
    rl [pref] : A . P => {A}P .

    *** Summation
    crl [sum] : P + Q => {A}P'  if  P => {A}P' .

    *** Composition
    crl [par] : P | Q => {A}(P' | Q)  if  P => {A}P' .
    crl [par] : P | Q => {tau}(P' | Q')
         if  P => {L}P'  /\  Q => {~ L}Q' .

    *** Restriction
    crl [res] : P \ L => {A}(P' \ L)
         if  P => {A}P' /\  (A =/= L)  /\  (A =/= ~ L) .

    *** Relabelling
    crl [rel] : P[M / L] => {M}(P'[M / L]) if P => {L}P' .
    crl [rel] : P[M / L] => {~ M}(P'[M / L]) if P => {~ L}P' .
    crl [rel] : P[M / L] => {A}(P'[M / L])
         if P => {A}P' /\  (A =/= L)  /\  (A =/= ~ L) .

    *** Definition
    crl [def] X => {A}P'
```

```
          if (X definedIn context)  /\  def(X, context) => {A}P' .
endm
```

This representation of CCS in Maude is semantically correct in the sense that given a CCS process P, there are processes $P_1, \ldots, P_{k-1}$ such that

$$P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{k-1}} P_{k-1} \xrightarrow{a_k} P'$$

if and only if `P` can be rewritten into `{a1}...{ak}P'` (see [41]).

A rewrite theory has both rules and equations, so that rewriting is performed *modulo* such equations. However, this does not mean that the Maude implementation must have a matching algorithm for each equational theory that a user might specify. In fact, this is impossible, since matching modulo an arbitrary theory is undecidable. The proposed solution is to divide the equations E into a set A of axioms, for which matching algorithms exist in the Maude implementation, [3] and a set E' of equations that are Church-Rosser, terminating, and sort decreasing *modulo* A; that is, the equational part must satisfy the same requirements as a functional module.

Moreover, we require that the rules R in the module are *coherent* [67] (or at least what might be called "weakly coherent" [44,68]) with the equations E' modulo A. This means that appropriate critical pairs between rules and equations are joinable, allowing us to intermix rewriting with rules and rewriting with equations without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken. In this way, we get the effect of rewriting modulo $E' \cup A$ with just a matching algorithm for A.

Under these circumstances, the *default strategy* in the Maude interpreter applies the rules in a top-down rule fair way, [4] always reducing to canonical form using E' before applying any rule in R. More specifically, before the application of each rewrite rule, the expression is simplified to its canonical form by applying the equations E' modulo A; then, the rule is applied to such a simplified expression modulo the axioms A according to the default strategy.

[3] Maude's rewrite engine has an extensible design, so that matching algorithms for new theories can be added and can be combined with existing ones [32]. As already mentioned, matching modulo associativity, commutativity, and (left-, right- or two-sided) identity, and combinations of these attributes are supported.

[4] "Top-down" means that each rewrite is attempted beginning at the top of the term, so that any position rewritten does not have a position above it that could also have been rewritten. A limited form of fairness is achieved by keeping the rules in a circular list, and moving a rule to the end of the list after it has been applied.

As another example of a system module, we specify a simple concurrent system, the blocks world, a typical example in artificial intelligence circles. In this version there is a table on top of which we have the blocks, which can be moved by means of three actions. A block is represented as a record with three fields: a label identifying the block (given by a quoted identifier, as provided in the predefined module `QID`), the label of the block on top (or the constant `clear` if there is none), and the label of the block below (or the constant `table` if there is none because the block is on the table). A state of the blocks world is then represented as a *set* of such blocks that is *consistent* in the sense that each block has a different label, and that for each pair of blocks a and b, if a is on top of b, then b is below a. In the module below we only make explicit the first part of the consistency check (all block labels are different).

```
mod BLOCKS-WORLD is
    protecting QID .
    sorts Up Down .
    subsorts Qid < Up Down .
    op clear : -> Up [ctor] .
    op table : -> Down [ctor] .
    sort Block .
    op {label:_, under:_, on:_} : Qid Up Down -> Block [ctor] .

    sort State .
    subsort Block < State .
    op empty : -> State [ctor] .
    op __ : State State -> [State] [ctor assoc comm id: empty] .
    op free : Qid State -> Bool .

    vars X Y Z : Qid .      vars S S' : State .
    var U U' : Up .         vars O O' : Down .

    cmb {label: X, under: U, on: O} S : State if free(X, S) .
    eq free(X, empty) = true .
    ceq free(X, S) = X =/= Y and free(X, S')
         if {label: Y, under: U, on: O} S' := S .

    rl [move] :     {label: X, under: clear, on: Z}
                    {label: Z, under: X,     on: O}
                    {label: Y, under: clear, on: O'}
                 => {label: X, under: clear, on: Y}
                    {label: Z, under: clear, on: O}
                    {label: Y, under: X,     on: O'} .

    rl [unstack] : {label: X, under: clear, on: Z}
```

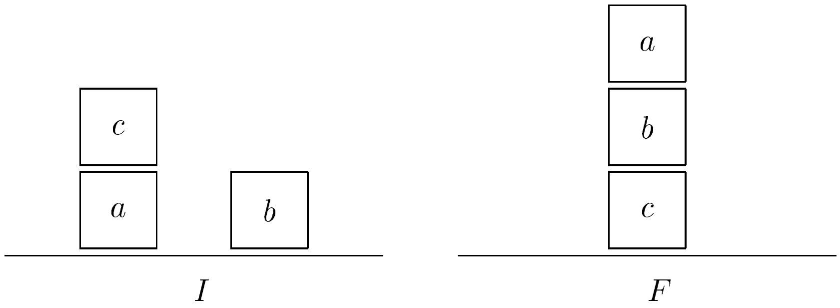

Fig. 1. Initial and final states in a world with three blocks.

```
                {label: Z, under: X,     on: O}
             => {label: X, under: clear, on: table}
                {label: Z, under: clear, on: O} .

    rl [stack] :   {label: X, under: clear, on: table}
                {label: Z, under: clear, on: O}
             => {label: X, under: clear, on: Z}
                {label: Z, under: X,     on: O} .
endm
```

The rule `move` moves a block `X` sitting on top of another block `Z` to the top of block `Y`. The rule `unstack` moves a block `X` sitting on top of another block `Z` to the table, whereas the rule `stack` does the reverse action.

Consider for example the states described in Figure 1. The initial state I on the left and the final state F on the right are respectively described by the following two terms of sort `State`:

```
{label: 'a, under: 'c,    on: table}
{label: 'c, under: clear, on: 'a}
{label: 'b, under: clear, on: table}

{label: 'c, under: 'b,    on: table}
{label: 'b, under: 'a,    on: 'c}
{label: 'a, under: clear, on: 'b}
```

The fact that the "sequential plan" (in a self-explanatory intuitive notation) $unstack(c, a); stack(b, c); stack(a, b)$ moves the blocks from state I to state F corresponds directly to a sequence of computational rewrite steps applying the corresponding rewrite rules.

3.4 Object-Oriented Modules

Among the many concurrent systems that we can specify as system modules in Maude, concurrent object-oriented systems are an important subclass [44].

In a concurrent object-oriented system the concurrent state, which is usually called a *configuration*, has typically the structure of a multiset made up of objects and messages that evolves by concurrent rewriting modulo associativity, commutativity and identity, using rules that describe the effects of *communication events* between objects and messages.

An *object* in a given state is represented in Maude as a term

$$\texttt{<}\ O : C\ \texttt{|}\ a_1 : v_1\ \texttt{,}\ \ldots\ \texttt{,}\ a_n : v_n\ \texttt{>}$$

where O is the object's name or identifier, C is its class identifier, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*. *Messages* do not have a fixed syntactic form; such syntactic form is defined by the user for each application. The concurrent state of an object-oriented system is then a multiset of objects and messages, called a `Configuration`, with multiset union described with empty syntax `__`.

The following module `CONFIGURATION` defines the basic concepts of concurrent object systems. Note that the sorts `Msg` and `Attribute`, as well as the sorts `Oid` and `Cid` of object and class identifiers, are left unspecified. They will become fully defined when the `CONFIGURATION` module is extended by specific object-oriented definitions in a given object-oriented module.

```
fmod CONFIGURATION is
    sorts Oid Cid Attribute AttributeSet
          Object Msg Configuration .
    subsorts Object Msg < Configuration .
    subsort Attribute < AttributeSet .
    op none : -> AttributeSet [ctor] .
    op _,_ : AttributeSet AttributeSet -> AttributeSet
         [ctor assoc comm id: none] .
    op <_:_|_> : Oid Cid AttributeSet -> Object [ctor] .
    op none : -> Configuration [ctor] .
    op __ : Configuration Configuration -> Configuration
         [ctor assoc comm id: none] .
endfm
```

Concurrent object-oriented systems are defined in Maude by means of *object-oriented modules*—introduced by the keyword `omod`—using a syntax more convenient than that of system modules because it assumes acquaintance with the basic entities, such as objects, messages, and configurations, and supports linguistic distinctions appropriate for the object-oriented case. In particular, all object-oriented modules implicitly include the above `CONFIGURATION` module and assume its syntax.

Classes are defined with the keyword `class`, followed by the name of the class

C, and by a list of attribute declarations separated by commas. Each attribute declaration has the form $a : S$, where a is an attribute identifier and S is the sort in which the values of the attribute range; that is, class declarations have the form `class` C `|` a_1`:` S_1`,` ... `,` a_n`:` S_n `.`

The rewrite rules in an object-oriented module specify in a declarative way the behavior associated with the messages. The multiset structure of the configuration provides the top-level distributed structure of the system and allows concurrent application of the rules [44].

By convention, the only object attributes made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only on the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only on the righthand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged.

The following object-oriented module gives an object-oriented specification of the blocks world described in Section 3.3. A block is now represented as an object with two attributes, `under`, saying whether it is under another block or it is clear, and `on`, saying whether the block is on top of another block or is on the table.

```
omod OO-BLOCKS-WORLD is
    protecting QID .
    sorts BlockId Up Down .
    subsorts Qid < BlockId < Oid .
    subsorts BlockId < Up Down .
    op clear : -> Up [ctor] .
    op table : -> Down [ctor] .
    class Block | under : Up, on : Down .
    vars X Y Z : BlockId .

    rl [move] :     < X : Block | under : clear, on : Z >
                    < Z : Block | under : X >
                    < Y : Block | under : clear >
                 => < X : Block | on : Y >
                    < Z : Block | under : clear >
                    < Y : Block | under : X > .

    rl [unstack] : < X : Block | under : clear, on : Z >
                   < Z : Block | under : X >
                => < X : Block | on : table >
                   < Z : Block | under : clear > .

    rl [stack] :   < X : Block | under : clear, on : table >
                   < Z : Block | under : clear >
                => < X : Block | on : Z >
```

```
                    < Z : Block | under : X > .
endom
```

The states I and F in Figure 1 are respectively described now by the following two configurations:

```
    < 'a : Block | under : 'c,    on : table >
    < 'c : Block | under : clear, on : 'a >
    < 'b : Block | under : clear, on : table >

    < 'c : Block | under : 'b,    on : table >
    < 'b : Block | under : 'a,    on : 'c >
    < 'a : Block | under : clear, on : 'b >
```

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'` in an object-oriented module is just a particular case of a subsort declaration. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses as well as the newly defined attributes, messages, and rules of the subclass characterize the structure and behavior of the objects in the subclass.

Suppose that the blocks world is further refined so that now blocks can have colors, but we still want the rules for manipulating blocks to remain the same. This is trivially achieved by class inheritance as illustrated by the following module.

```
omod OO-BLOCKS-WORLD+COLOR is
    including OO-BLOCKS-WORLD .
    sort Color .
    ops red blue yellow : -> Color [ctor] .
    class ColoredBlock | color : Color .
    subclass ColoredBlock < Block .
endom
```

In this example, there is only one class immediately above `ColoredBlock`, namely, `Block`, but a class may in general be defined as a subclass of several classes, i.e., *multiple inheritance* is also supported. If an attribute and its sort have already been declared in a superclass, they should not be declared again in the subclass; indeed, all such attributes are *inherited.* In the case of multiple inheritance, when an attribute occurs in two different superclasses, then the sort associated to it in each of those superclasses must be the same. Then, a class inherits all the attributes, messages, and rules from all its superclasses. An object in the subclass behaves exactly as any object in any of the superclasses, but it may exhibit additional behavior due to the introduction of new attributes, messages, and rules in the subclass.

The semantics of object-oriented modules is entirely reducible to that of system modules, in the sense that each object-oriented module can be translated into a corresponding system module whose semantics *is* by definition that of the original object-oriented module [44,27]. In particular, rewrite rules are modified to make them applicable to all objects of the given classes and of their subclasses, that is, not only to objects whose class identifiers are those explicitly given.

However, although Maude's object-oriented modules are in this way reduced to system modules, there are of course important conceptual advantages provided by the syntax of object-oriented modules. This syntax allows the user to think and express his or her thoughts in object-oriented terms whenever such a viewpoint seems best suited for the problem at hand. Those conceptual advantages would be lost if only system modules were provided. For example, in an object-oriented configuration we have objects that maintain their *identity* across their state changes, and the notions of fairness adequate for them are more specialized than those appropriate for arbitrary system modules. This is because, since each object has an individual identity, fairness should now be *localized* to individual objects and messages, which should not be *starved* even when other similar objects and messages are rewritten.

In summary, the approach taken in Maude is to provide a logical semantics for concurrent object-oriented programming by taking rewriting logic as its foundation, and then defining in a rigorous way higher-level object-oriented concepts above such a foundation. The papers [44,45] provide good background on such foundations. Talcott's papers [62–65] give rewriting logic foundations for actors from a somewhat different viewpoint. The paper [53] shows how, for object-oriented modules satisfying some simple requirements, their initial model semantics coincides with a very natural truly concurrent semantics based on a partial order of events.

One important strength of the object-oriented viewpoint is that all kinds of entities in the external world can be conceptualized as objects and can be interacted with from a computation by message passing. Built-in objects extend Maude with interfaces allowing interaction with external entities such as internet sockets, file systems, window systems, and so on. In this way, the computation can be connected with the external world and with other Maude computations in different machines in a distributed way. Interfaces to external entities are specified by means of *built-in object-oriented modules* defining built-in objects.

Such built-in object-oriented modules can be imported by ordinary object-oriented modules so that, in general, the object-oriented state of a computation consists of two parts: a configuration of ordinary objects and messages that is represented in Maude as a multiset of terms representing such objects and messages, and a set of built-in objects, together with messages to and from those objects. Conceptually we can think of these two parts as a single bigger configuration of objects and messages. However, built-in objects are not themselves visible in the configuration of ordinary objects and messages, except indirectly, through the messages that they send. In particular, the internal structure of built-in objects is hidden, so that they can only be interacted with by asynchronous message passing.

4 Module Operations and Parameterized Programming

Specifications and code should be structured in *modules* of relatively small size to facilitate understandability of large systems, increase reusability of components, and localize the effects of system changes. Maude fully supports these goals by means of a rich and extensible *module algebra* supporting, in particular, parameterized programming techniques in the OBJ3 style [37]. Moreover, Maude provides useful basic support for modularity by allowing the definition of *module hierarchies*, that is, acyclic graphs of module importations.

Parameterized modules, theories, and views are the basic building blocks of parameterized programming [10,37,26,23]. As in OBJ, a theory [5] defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter. The instantiation of the formal parameters of a parameterized module with actual parameter modules requires a view from the formal interface theory to the corresponding actual module. That is, views provide the interpretation of the actual parameters. For more details on parameterized modules in Maude, the reader is advised to consult [27].

4.1 Module Hierarchies

Mathematically, we can think of module hierarchies as partial orders of *rewrite theory inclusions*, that is, the theory of the importing module contains the theories of its submodules as subtheories. Recall that a *rewrite theory* is a four-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic. As already explained in Section 3.2, a system module is a rewrite theory with *initial semantics*. Note that we can use the inclusion of membership equational logic into rewriting logic to view a functional module specifying an equational theory (Ω, E) as a degenerate case of a rewrite theory, namely the rewrite theory $(\Omega, E, \emptyset, \emptyset)$. In fact the initial algebra of (Ω, E) and the

[5] The reader should be careful in not confusing the different uses of the word "theory" in this section.

initial model of $(\Omega, E, \emptyset, \emptyset)$ coincide [43]. Therefore, in essence we can view all modules as rewrite theories.

The most general form of module inclusion is provided by the `including` keyword, followed by the name of the imported module. The `protecting` keyword is a more restricted form of inclusion, in the sense that it makes a *semantic assertion* about the relationship between the initial models of the two theories. Let $\mathcal{R} = (\Omega, E, L, R)$ be the rewrite theory specified by a system module, and let $\mathcal{R}' = (\Omega', E', L', R')$ be the theory of a supermodule, so that we have a theory inclusion $\mathcal{R} \subseteq \mathcal{R}'$. Then, we can view each model $\mathcal{M}'$ of $\mathcal{R}'$ as a model $\mathcal{M}'|_{\mathcal{R}}$ of $\mathcal{R}$, simply by disregarding the extra sorts, operators, equations, membership axioms, and rules in $\mathcal{R}' - \mathcal{R}$. Since, as explained in Section 3.2, the rewrite theories $\mathcal{R}$ and $\mathcal{R}'$ have respective initial models $\mathcal{T}_{\mathcal{R}}$ and $\mathcal{T}_{\mathcal{R}'}$, by initiality of $\mathcal{T}_{\mathcal{R}}$ we always have a unique $\mathcal{R}$-homomorphism $h : \mathcal{T}_{\mathcal{R}} \longrightarrow \mathcal{T}_{\mathcal{R}'}|_{\mathcal{R}}$.

In the models of a rewrite theory the sorts are interpreted as categories. Then, the `protecting` importation asserts that for each sort s in the signature Ω of $\mathcal{R}$ the function h_s is an isomorphism of categories. Intuitively, this means that the initial model of the supermodule does not add any "junk" or any "confusion" to the initial model of the submodule. Note that the expected condition would have been to require h to be an $\mathcal{R}$-*isomorphism*. However, due to the presence of error elements at the kind level, the isomorphism condition would be too strong, since in general, when enlarging a signature, there will be new error terms that cannot be proved equal to old ones. See [9] for a detailed discussion of, and proof techniques for, protecting extensions in membership equational logic.

Of course, the `protecting` assertion cannot be checked by Maude at runtime. It requires inductive theorem proving. Using the proof techniques in [9] together with an inductive theorem prover for membership equational logic and a Church-Rosser checker such as those described in [17], this can be done for functional modules; and it seems natural to expect that these techniques and tools will extend to similar ones for rewrite theories.

By contrast, the `including` assertion does not make such requirements on h. It does, however, make *some* requirements. Namely, if the subtheory $\mathcal{R}$ does itself contain a proper subtheory $\mathcal{R}_0$ that it imports in `protecting` mode, then the inclusion $\mathcal{R}_0 \subseteq \mathcal{R}'$ is still assumed to be `protecting`. For such an inclusion to become an `including` assertion, we have to say so by explicitly listing the module defining $\mathcal{R}_0$ in the list of modules imported in `including` mode.

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports three different types of theories: functional theories, system theories, and object-oriented theories. Their structure is the same as that of their module counterparts.

Theories are rewriting logic theories with a *loose interpretation*, as opposed to modules that have an initial semantics. Therefore, theories are allowed to contain more general sentences that need not satisfy all the requirements described for modules.

Let us begin by introducing the functional theory `TRIV`, which requires just a sort.

```
fth TRIV is
    sort Elt .
endfth
```

The theory of partially ordered sets with an antireflexive and transitive binary operator is expressed in the following way. [6]

```
fth POSET is
    protecting BOOL .
    sort Elt .
    op _<_ : Elt Elt -> Bool .
    vars X Y Z : Elt .
    eq X < X = false .
    ceq X < Z = true if X < Y and Y < Z .
endfth
```

The theory of totally ordered sets, that is, posets in which all pairs of distinct elements have to be related, is specified as follows:

```
fth TOSET is
    including POSET .
    vars X Y : Elt .
    eq X < Y or Y < X or X == Y = true .
endfth
```

The `including` importation of a theory into another theory keeps its loose semantics. However, if the imported theory contains a module, which therefore

[6] As with modules, theories implicitly import the predefined module `BOOL`, and therefore the `protecting BOOL` declaration is unnecessary.

must be interpreted with an initial semantics,[7] then that initial semantics is maintained by the importation. For example, in the definition of the POSET theory, the declaration protecting BOOL ensures that the initial semantics of the functional module for the Booleans is preserved, which is in fact a crucial requirement. This requirement is then preserved by TOSET when POSET is included.

4.3 Parameterized Modules

Theories are used to declare the interface requirements for parameterized modules. Modules can be parameterized by one or more theories. All theories appearing in the interface must be labelled in such a way that their sorts can be uniquely identified. The general form for the interface of a parameterized module is $(X_1 :: T_1, \ldots, X_n :: T_n)$ where $X_1, \ldots, X_n$ are the labels and $T_1, \ldots, T_n$ are the names of the respective parameter theories.

All the sorts coming from theories in the interface must be qualified by their labels, even if there is no ambiguity. If Z is the label of a parameter theory T, then each sort S in T has to be qualified as $Z.S$ (the reason for this will be explained below). Moreover, there cannot be subsort overloading between an operator declared in a theory being used as parameter of a parameterized module and an operator declared in the body of the parameterized module, or between operators declared in two parameter theories of the same module.

In the body of a parameterized module $M(X_1 :: T_1, \ldots, X_n :: T_n)$, any parameterized sort S is written in the form $S(X_1, \ldots, X_n)$. When the module is instantiated with views $V_1, \ldots, V_n$ then this sort becomes $S(V_1, \ldots, V_n)$. Thus, a simple parameterized module for lists is defined as follows:

```
fmod LIST(X :: TRIV) is
    sort List(X) .
    subsort X.Elt < List(X) .
    op nil : -> List(X) [ctor] .
    op __ : List(X) List(X) -> List(X) [ctor assoc id: nil] .
endfm
```

The module LIST has only one parameter. In general, as already mentioned, parameterized modules can have several parameters. It can furthermore happen that several parameters are declared with the same parameter theory. Therefore, parameters cannot be treated as normal submodules, since we do not want them to be shared when their labels are different. We regard the

[7] In Maude, the importation of a module into a theory is supported only in protecting mode.

relationship between the body of a parameterized module and the interface of its parameters not as an inclusion, but as a module constructor which is evaluated generating renamed copies of the parameters, which are then included. In such copies of parameter theories sorts are renamed as follows: If `Z` is the label of a parameter theory `T`, then each sort `S` in `T` is renamed to `Z.S`. This is the reason why all occurrences of these sorts in the body of the parameterized module must mention their corresponding renaming, as explained before.

Let us consider as an example the following module `TUPLE[2]`. Notice the use of the qualifications for the sorts coming from each of the parameters, and notice also the form of the sort `Tuple(C1, C2)`.

```
fmod TUPLE[2](C1 :: TRIV, C2 :: TRIV) is
    sort Tuple(C1, C2) .
    op ((_,_)) : C1.Elt C2.Elt -> Tuple(C1, C2) [ctor] .
    op p1_ : Tuple(C1, C2) -> C1.Elt .
    op p2_ : Tuple(C1, C2) -> C2.Elt .
    var E1 : C1.Elt .
    var E2 : C2.Elt .
    eq p1 (E1, E2) = E1 .
    eq p2 (E1, E2) = E2 .
endfm
```

In Maude, the module expression `TUPLE[`n`]`, for n a nonzero natural number, generates a parameterized module specifying a tuple of the corresponding size. For example, for n equal to 2, the system generates automatically the parameterized module `TUPLE[2]` given above.

4.4 Views

Views are used to assert how a particular target module or theory is claimed to satisfy a source theory. In general, there may be several ways in which such requirements might be satisfied, if at all, by the target module or theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target. Each view declaration has an associated set of *proof obligations*, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques of the style supported for Maude's logic in [17].

All views have to be defined explicitly, and all of them must have a name. As any theory or module, views should have been defined before they are used. In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort, operator, class, and message in the source theory, although it is possible to simplify such mappings (see [27]).

The following view shows how MACHINE-INT satisfies the theory TRIV:

```
view Int from TRIV to MACHINE-INT is
    sort Elt to MachineInt .
endv
```

We can also have views between theories, such as the following:

```
view Toset from TRIV to TOSET is
    sort Elt to Elt .
endv
```

Moreover, views can be parameterized:

```
view Tuple(X :: TRIV, Y :: TRIV) from TRIV to TUPLE[2](X, Y) is
    sort Elt to Tuple(X, Y) .
endv
```

Note that the view Tuple is parameterized by two different instances of the theory TRIV. Parameterized views of this kind allow us to keep the parameter part of the target uninstantiated. The paper [31] discusses the use of parameterized theories and views in Maude.

4.5 Module Instantiation

Instantiation is the process by which actual parameters are bound to the parameters of a parameterized module and a new module is created as a result. This can be seen in fact as the evaluation of a module expression. The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the formal parameters to the corresponding sorts, operators (or expressions), etc. in the target.

A parameterized module is instantiated with views explicitly defined previously. For example, we can define a module providing finite lists of pairs, whose first components are machine integers and whose second components are still parameterized by means of the module expression LIST(Tuple(Int, X)), which uses the view Int as well as an instance of the parameterized view Tuple, both defined in Section 4.4. This expression is used in the following module, which is a general parameterized version of the array representation module in Section 2.3.

```
fmod ARRAY(X :: TRIV) is
    protecting LIST(Tuple(Int, X)) .
    sorts NeArray(X) Array(X) .
    subsorts Tuple(Int, X) < NeArray(X)
                < Array(X) < List(Tuple(Int, X)) .
    op _[_] : NeArray(X) MachineInt -> [X.Elt] .
    op _[_->_] : NeArray(X) MachineInt X.Elt -> [Array(X)] .
    ops low high : NeArray(X) -> MachineInt .

    vars I J : MachineInt .
    vars Z Y : X.Elt .
    vars L L' : List(Tuple(Int, X)) .

    mb nil : Array(X) .
    cmb (I, Z) (J, Y) L : NeArray(X)
       if (I + 1 = J) /\ (J, Y) L : NeArray(X) .

    ceq (L (I, Z) L')[I] = Z if L (I, Z) L' : NeArray(X) .
    ceq (L (I, Z) L')[I -> Y] = (L (I, Y) L')
       if L (I, Z) L' : NeArray(X) .

    ceq low((I, Z) L) = I if (I, Z) L : NeArray(X) .
    ceq high(L (I, Z)) = I if L (I, Z) : NeArray(X) .
endfm
```

As mentioned in Section 4.4, we can define views from theories to theories and can use such views to define new parameterized modules. For example, we can define a parameterized system module specifying a sorting rule on arrays whose elements belong to a totally ordered set as follows:

```
mod SORTING(X :: TOSET) is
    protecting ARRAY(Toset)(X) .
    vars I J : MachineInt .
    vars Z Y : X.Elt .
    var L : List(Tuple(Int, Toset))(X) .
    crl [sort] : (I, Z) L (J, Y) => (I, Y) L (J, Z)
       if Z > Y  /\  (I, Z) L (J, Y) : NeArray(Toset)(X) .
endm
```

The module `INT-SORTING` in Section 3.2 can be obtained as the module expression `SORTING(IntAsToset)` where

```
view IntAsToset from TOSET to MACHINE-INT is
    sort Elt to MachineInt .
    vars X Y : Elt .
    op X < Y to term X <= Y and X =/= Y .
```

```
endv
```

Note that an operator can be mapped to a term. In the `IntAsToset` view, for illustration purposes, the `_<_` relation of a toset is mapped to an expression using the "less than or equal" operator `_<=_` and the inequality operator `_=/=_` in `MACHINE-INT`, instead of using directly the operator `_<_` in `MACHINE-INT`.

5 Reflection and the `META-LEVEL`

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully represented in itself. Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective [12,20]. This makes the metatheory of rewriting logic accessible to the user in a clear and principled way. However, since a naive implementation of reflection can be computationally expensive, a good implementation must provide efficient ways of performing reflective computations. This section explains how this is achieved in Maude through its predefined `META-LEVEL` module.

5.1 Reflection and Metalevel Computation

Rewriting logic is reflective in a precise mathematical way, namely, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}, \overline{t'}$, and any pair $(\mathcal{R}, t)$ as a term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, in such a way that we have the following equivalence

$$(\dagger)\ \ \mathcal{R} \vdash t \longrightarrow t' \ \ \Leftrightarrow \ \ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, because we have

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

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In a naive implementation, each step up the reflective tower comes at considerable computational cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible—so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary.

To achieve a systematic descent into equivalent rewriting computations at lower levels, the key idea is to exploit the equivalence (†). Detailed proofs of this equivalence have been given for unsorted unconditional theories [12] and for unsorted and many-sorted conditional theories [20]. The extension to the case of interest for Maude—namely to conditional rewrite theories with membership equational logic as the underlying equational logic—although nontrivial, is essentially unproblematic. We therefore assume a universal theory $\mathcal{U}$ for this more general class of finitely presented rewrite theories. In particular, the signature $\Sigma_{\mathcal{U}}$ of $\mathcal{U}$ has sorts *Term*, *Module*, and *Kind*, whose respective elements $\overline{t} : \mathit{Term}$, $\overline{\mathcal{R}} : \mathit{Module}$, and $\overline{K} : \mathit{Kind}$ represent terms, rewrite theories, and kinds in a signature, respectively. We assume that there is also an equationally defined Boolean predicate $\mathit{parse} : \mathit{Module} \times \mathit{Kind} \times \mathit{Term} \longrightarrow \mathit{Bool}$ so that $\mathit{parse}(\overline{\mathcal{R}}, \overline{K}, \overline{t}) = \mathit{true}$ if t is an $\mathcal{R}$-term of kind K, and $\mathit{parse}(\overline{\mathcal{R}}, \overline{K}, \overline{t}) = \mathit{false}$ otherwise.

We can exploit the equivalence (†) by introducing the notion of *descent function*, that is, a function that, given metalevel representations for a rewrite theory $\mathcal{R}$ and a term t in it, rewrites such a term in $\mathcal{R}$ according to a given strategy and returns the metarepresentation of the resulting term. Such functions can be simply expressed in terms of a general *sequential interpreter function* I for rewriting logic. This is a *partial* function that takes three arguments: a finitely presented rewrite theory $\mathcal{R}$, a term t, and a deterministic strategy S. In case of termination it returns either the term t' to which t was rewritten according to S, or an error message that is not a term in $\mathcal{R}$. The function is undefined in case the strategy does not terminate. For any finitely presented rewrite theory $\mathcal{R}$, terms t, t' in it, and admissible deterministic strategy S, any such interpreter function must of course satisfy the correctness requirement

$$(\flat)\ \ I(\mathcal{R}, t, S) = t' \ \ \Rightarrow \ \ \mathcal{R} \vdash t \longrightarrow t'.$$

The point is that, regardless of the particular details of I, we can always equationally axiomatize any such effective interpreter function by means of a Church-Rosser, but in general nonterminating, finitary equational theory $\mathcal{I}$. This can be done in a signature that we can assume contains $\Sigma_{\mathcal{U}}$ as a subsignature. By extending our universal theory $\mathcal{U}$ with the new sorts, operators, and equations of $\mathcal{I}$, we obtain an extended rewrite theory $\mathcal{U} \cup \mathcal{I}$. A *descent function* is then a function $d : \mathit{Module} \times \mathit{Term} \times \mathit{Parameters} \longrightarrow \mathit{Term}$ such that there is a deterministic strategy expression S_d with a single free variable of sort *Parameters* satisfying the equality $d(\overline{\mathcal{R}}, \overline{t}, p) = \overline{I(\mathcal{R}, t, S_d(p))}$.

Such descent functions are definable equationally as definitional extensions of the theory $\mathcal{U} \cup \mathcal{I}$. Note that, since we have only added some new equations, the only rewrite rules in $\mathcal{U} \cup \mathcal{I}$ are exactly those in $\mathcal{U}$. But, given a descent function d, we can now exploit the equivalence (†) by adding to $\mathcal{U} \cup \mathcal{I}$ a *descent rule*

$$
\begin{array}{l}
d : \langle M, x \rangle \longrightarrow \langle M, y \rangle \\
\quad \textit{if } \textit{parse}(M, K, x) = \textit{true} \wedge \textit{parse}(M, K, y) = \textit{true} \wedge d(M, x, p) = y.
\end{array}
$$

where M : *Module*, x, y : *Term*, K : *Kind*, and p : *Parameters*. The equivalence (†) can be exploited for efficiency reasons with such a rule, because the sequential interpreter I can be a built-in function such as the Maude interpreter; therefore, instantiating M with $\overline{\mathcal{R}}$, we can use efficient deduction in $\mathcal{R}$ to perform deduction in $\mathcal{U}$. Let $\mathcal{M}$ denote a rewrite theory of the form $\mathcal{M} = \mathcal{U} \cup \mathcal{I} \cup \mathcal{D}$, where $\mathcal{D}$ is the addition of several descent functions and of their associated descent rules. We shall call $\mathcal{M}$ a *metalevel theory*.

The addition of descent rules to $\mathcal{U}$ is of course *conservative*, in the sense of not adding any rewrites that could not be performed, albeit less efficiently, in $\mathcal{U}$ itself, since for any descent rule d we have

$$
\begin{aligned}
\mathcal{M} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \stackrel{d}{\longrightarrow} \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Rightarrow & \, I(\mathcal{R}, t, S_d(p)) = t' \\
\stackrel{\flat}{\Rightarrow} & \, \mathcal{R} \vdash t \longrightarrow t' \\
\stackrel{\dagger}{\Leftrightarrow} & \, \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.
\end{aligned}
$$

Note that, by applying several descent functions, we can descend several levels in the reflective tower; that is, a meta-metalevel computation can be efficiently carried out at the object level. More generally, we should view descent functions as *basic strategies*, that can be used as fundamental building blocks to define *internal strategy languages*, in which they can be combined with each other and with more complex strategies at several levels of reflection to perform efficiently sophisticated metalevel computations (see Section 6).

5.2 The Module `META-LEVEL`

In Maude, key functionality of a metalevel theory $\mathcal{M}$ with several descent functions has been efficiently implemented in a functional module `META-LEVEL`, by using as the interpreter function I Maude's own interpreter. Furthermore, several other useful functions of the universal theory $\mathcal{U}$ are also built-in for efficiency reasons. In the module `META-LEVEL`:

- Maude terms are reified as elements of a data type `Term` of terms;
- Maude modules are reified as terms in a data type `Module` of modules;
- the processes of reducing a term to normal form in a functional module and of finding whether such a normal form has a given sort are reified by a descent function `metaReduce`;
- the process of applying a rule of a system module to a subject term is reified by descent functions `metaApply` and `metaXapply`;
- the process of rewriting a term in a system module using Maude's default interpreter is reified by a descent function `metaRewrite`;
- the process of matching a pattern to a subject term is reified by descent functions `metaMatch` and `metaXmatch`; and
- parsing and pretty printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also reified by corresponding metalevel functions.

Sorts and kinds are represented as specific subsorts of the sort `Qid` of quoted identifiers. Since operator declarations can use both sorts and kinds, we denote by `Type` the union of `Sort` and `Kind`.

```
subsorts Sort Kind < Type < Qid.
subsort Type < TypeList .
```

5.3 Representing Terms

Terms are reified as elements of the data type `Term` of terms. The basic cases in the representation of terms are obtained by subsorts `Constant` and `Variable` of the sort `Qid`. Constants are quoted identifiers that contain the constant's name and its type separated by a "`.`", e.g., `'0.Nat`. Similarly, variables contain their name and type separated by a "`:`", e.g., `'N:Nat`. Appropriate selectors extract their names and types.

```
subsorts Constant Variable < Qid .
op getName : Constant -> Qid .    op getName : Variable -> Qid .
op getType : Constant -> Type .   op getType : Variable -> Type .
```

Then a term is constructed in the usual way, by applying an operator symbol to a list of terms.

```
subsorts Constant Variable < Term .
op _[_] : Qid TermList -> Term [ctor] .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [ctor assoc] .
```

Since terms in the module META-LEVEL can be metarepresented just as terms in any other module, the representation of terms can be iterated. For example, the term s (N:Nat) + 0 in the module NAT in Section 5.4, specifying natural numbers in Peano notation, is metarepresented by

```
'_+_['s_['N:Nat], '0.Nat],
```

and meta-metarepresented by

```
'_`[_`]['`'_+_.Qid, '_`,_['_`[_`]['`'s_.Qid, '`'N:Nat.Variable],
                            '`'0.Nat.Constant]] .
```

5.4 Representing Modules

Functional and system modules are metarepresented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are now metarepresented as we have already explained in the previous section; (2) in the metarepresentation of modules we follow a fixed order in introducing the different kinds of declarations for sorts, subsort relations, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; (3) there is no need for variable declarations; and (4) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operator _;_.

The syntax for the top-level operators representing functional and system modules is as follows:

```
sorts FModule Module .
subsort FModule < Module .

op fmod_is_sorts_.____endfm : Qid ImportList SortSet SubsortDeclSet
     OpDeclSet MembAxSet EquationSet -> FModule [ctor] .

op mod_is_sorts_._____endm : Qid ImportList SortSet SubsortDeclSet
     OpDeclSet MembAxSet EquationSet RuleSet -> Module [ctor] .
```

Without going into all the syntactic details, we show only the operators used to represent conditions, equations, and rules.

```
sorts EqCondition Condition .
subsort EqCondition < Condition .
ops (_=_) (_:=_) : Term Term -> EqCondition [ctor] .
op _:_ : Term Sort -> EqCondition [ctor] .
```

```
  op _=>_ : Term Term -> Condition [ctor] .
  op _/\_ : EqCondition EqCondition -> EqCondition [ctor assoc] .
  op _/\_ : Condition Condition -> Condition [ctor assoc] .

  sorts Equation EquationSet .
  subsort Equation < EquationSet .
  op eq_=_. : Term Term -> Equation [ctor] .
  op ceq_=_if_. : Term Term EqCondition -> Equation [ctor] .
  op none : -> EquationSet [ctor] .
  op __ : EquationSet EquationSet -> EquationSet
       [ctor assoc comm id: none] .

  sorts Rule RuleSet .
  subsort Rule < RuleSet .
  op rl[_]:_=>_. : Qid Term Term -> Rule [ctor] .
  op crl[_]:_=>_if_. : Qid Term Term Condition -> Rule [ctor] .
  op none : -> RuleSet [ctor] .
  op __ : RuleSet RuleSet -> RuleSet [ctor assoc comm id: none] .
```

As a simple example, the metarepresentation of the module on the left is the term displayed on the right, so that the reader can appreciate the similarity between both notations:

```
fmod NAT is                   fmod 'NAT is
                                nil
  sorts Zero Nat .              sorts 'Zero ; 'Nat .
  subsort Zero < Nat .          subsort 'Zero < 'Nat .
  op 0 : -> Zero [ctor] .       op '0 : nil -> 'Zero [ctor] .
  op s_ : Nat -> Nat [ctor] .   op 's_ : 'Nat -> 'Nat [ctor] .
  op _+_ : Nat Nat -> Nat       op '_+_ : 'Nat 'Nat -> 'Nat
     [comm] .                       [comm] .
  vars N M : Nat .              none
  eq 0 + N = N .                eq '_+_['0.Nat, 'N:Nat] = 'N:Nat .
  eq (s N) + M = s (N + M) .    eq '_+_['s_['N:Nat], 'M:Nat]
                                  = 's_['_+_['N:Nat, 'M:Nat]] .
endfm                         endfm
```

Since `NAT` has no list of imported submodules and no membership axioms, those fields are filled, respectively, with the constants `nil` of sort `ImportList`, and `none` of sort `MembAxSet`.

Note that terms of sort `Module` can be metarepresented again, yielding then a term of sort `Term`, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels. A good example is the inductive theorem prover described in [17], where modules are metarepresented as terms of sort `Module` in the inference rules for induction, but they have to be meta-metarepresented as terms of sort `Term` when used in strategies that control the application of the inductive inference rules.

5.5 Descent Functions

The module META-LEVEL has several built-in descent functions that provide useful and efficient ways of reducing metalevel computations to object-level ones.

The operation metaReduce takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module.

```
op metaReduce : Module Term -> [ResultPair] .
op {_,_} : Term Type -> ResultPair [ctor] .
```

It returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, together with its corresponding sort or kind.

The interpreter function for $\texttt{metaReduce}(\overline{\mathcal{R}}, \overline{t})$ rewrites the term t to normal form using only the equations in $\mathcal{R}$, and does so according to the operator evaluation strategies (see the end of Section 2.2 and [33]) declared for each operator in the signature of $\mathcal{R}$, which by default is bottom-up for operators with no such strategies declared. In other words, the interpreter strategy for this function coincides with that of the reduce command in Maude, that is,

$$\texttt{metaReduce}(\overline{\mathcal{R}}, \overline{t}) = \overline{I_{Maude}(\mathcal{R}, t, \texttt{reduce})}.$$

The operation metaRewrite has syntax

```
op metaRewrite : Module Term MachineInt -> [ResultPair] .
```

It is entirely analogous to metaReduce, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term using Maude's default strategy. Its first two arguments are the representations of a module $\mathcal{R}$ and of a term t, and its third argument is a natural number n. Its result is the representation of the term obtained from t after at most n applications of the rules in $\mathcal{R}$ using the strategy of Maude's default interpreter, which applies the rules in a top-down rule fair way. When the value 0 is given as the third argument, no bound is given to the number of rewrites, and rewriting proceeds to the bitter end. Again, metaRewrite is a paradigmatic example of a descent function; its corresponding interpreter strategy is that of the rewrite command in Maude, that is,

$$\texttt{metaRewrite}(\overline{\mathcal{R}}, \overline{t}, n) = \overline{I_{Maude}(\mathcal{R}, t, \texttt{rewrite}\ [n])}.$$

The operation metaApply has syntax:

```
op metaApply : Module Term Qid Substitution MachineInt
             -> [ResultTriple] .
```

The first four arguments are representations in META-LEVEL of a module $\mathcal{R}$, a term t in $\mathcal{R}$, a label l of some rules in $\mathcal{R}$, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last argument is a natural number n used to enumerate all possible matches (due to the presence of structural axioms for operators or several rules with the same label l). metaApply then returns a triple of sort ResultTriple consisting of a term, with the corresponding sort or kind, and a substitution. The syntax for substitutions and for results is

```
subsort Assignment < Substitution .
op _<-_ : Qid Term -> Assignment [ctor] .
op none : -> Substitution [ctor] .
op _;_ : Substitution Substitution -> Substitution
     [ctor assoc comm id: none] .
op {_,_,_} : Term Type Substitution -> ResultTriple [ctor] .
```

The operation metaApply is evaluated as follows:

(1) the term t is first fully reduced using the equations in $\mathcal{R}$;
(2) the resulting term is matched against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded;
(3) the first n successful matches are discarded; if there is an $(n+1)$th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise an error is returned;
(4) the term resulting from applying the given rule with the $(n+1)$th match is fully reduced using the equations in $\mathcal{R}$;
(5) the triple formed using the constructor {_,_,_} whose first element is the representation of the resulting fully reduced term, whose second element is the representation of the corresponding type, and whose third element is the representation of the match used in the reduction is returned.

The interpreter strategy associated to metaApply$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \overline{\sigma}, n)$ is not that of a user-level command in the Maude interpreter. It is instead a built-in strategy internal to the interpreter that attempts one rewrite at the top as explained above.

The operation metaXapply, with syntax

```
op metaXapply : Module Term Qid Substitution MachineInt MachineInt
     MachineInt -> [Result4Tuple] .
op {_,_,_,_} : Term Type Substitution Context -> Result4Tuple [ctor] .
```

works as metaApply but using *matching with extension* (see [15, Section 5.8]) and in any possible position, not only at the top. The first two integer arguments indicate, respectively, the minimum and maximum depth in the flattened term (with respect to its associative or associative-commutative operators) where the application of the rule can take place. The last integer argument enumerates the solutions, since there can be different such rewrites with different substitutions and at different positions. The result has an additional component, giving the context inside the given term, where the rewriting has taken place. Contexts (terms with a single "hole") are defined as follows: [8]

```
subsort Context < CTermList .
subsorts TermList CTermList < GTermList .
op [] : -> Context [ctor] .
op _,_ : TermList CTermList -> CTermList [ctor assoc] .
op _,_ : CTermList TermList -> CTermList [ctor assoc] .
op _[_] : Qid CTermList -> Context [ctor] .
```

The function metaMatch intuitively tries to match at the top two given terms in a module. The last argument is used to enumerate possible matches. If the matching attempt is successful, the result is the corresponding substitution. The generalization to metaXmatch is analogous to the generalization to metaXapply.

```
op metaMatch : Module Term Term MachineInt -> [Substitution] .
op metaXmatch : Module Term Term MachineInt MachineInt
                MachineInt -> [MatchPair] .
op {_,_} : Substitution Context -> MatchPair [ctor] .
```

5.6 Parsing, Pretty Printing, and Sort Functions

Besides the descent functions already discussed, META-LEVEL provides several other functions that naturally belong to the universal theory and could have been equationally axiomatized in such a theory. However, for efficiency reasons they are provided as built-in functions. These functions allow parsing and pretty printing a term in a module at the metalevel, and performing efficiently a number of useful operations on the sorts declared in a module's signature.

The function metaParse takes as arguments the representation of a module, the representation of a list of tokens as a list of quoted identifiers, and, optionally, a sort or kind. It returns the metarepresentation of the parsed term

[8] Sort CTermList represents lists of terms with exactly a "hole" in the whole list, and sort GTermList is only needed for the assoc attribute, which is necessary, to make sense.

of that list of tokens for the signature of the module, which is assumed to be unambiguous.

The function `metaPrettyPrint` takes as arguments the representation of a module M and the representation of a term t. It returns a list of quoted identifiers that encode the string of tokens produced by pretty printing t in the syntax given by M. In the event of an error an empty list is returned.

The operations on sorts provide commonly needed functions on the poset of sorts of a module in a built-in way at the metalevel. For example, the function `leastSort` takes as arguments the representations of a module and a term and computes the (representation of the) least sort of that term in the module, while the Boolean expression $\texttt{sameKind(}\overline{M}\texttt{,}\overline{s}\texttt{,}\overline{s'}\texttt{)}$ is true if and only if the sorts s and s' belong to the same kind in the module M.

5.7 Extensions of `META-LEVEL`

In metalevel computations it is very convenient to be able to refer by name to the metarepresentations of modules already entered into the system. To make this possible, Maude allows importation declarations of the form

$$\texttt{protecting META-LEVEL(}M_1, \ldots, M_n\texttt{) .}$$

where $M_1, \ldots, M_n$ is a list of names of user-defined modules. With this declaration, new constants $M_1, \ldots, M_n$ of sort `Module` are declared, and new equations making each constant M_i equal to the metalevel representation of the module with name M_i (declared previously by the user) are added, for $i = 1 \ldots n$. Thus, after entering the module `NAT` in Section 5.4 above, we can declare a module that protects `META-LEVEL(NAT)` and defines a function to extract the set of operator declarations of a functional module as follows:

```
fmod META-NAT is
    protecting META-LEVEL(NAT) .
    op getOpDeclSet : FModule -> OpDeclSet .
    var QI : Qid .           var IL : ImportList .
    var SS : QidSet .        var SSDS : SubsortDeclSet .
    var ODS : OpDeclSet .    var MAS : MembAxSet .
    var EqS : EquationSet .
    eq getOpDeclSet(fmod QI is IL sorts SS . SSDS ODS MAS EqS endfm)
      = ODS .
endfm
```

Then we can apply this function to the constant `NAT`, which in `META-NAT` has been declared to be equal to the metarepresentation of the user-defined module

NAT, as follows:

```
Maude> red getOpDeclSet(NAT) .
Result OpDeclSet :
       op '0 : nil -> 'Zero [ctor] .
       op 's_ : 'Nat -> 'Nat [ctor] .
       op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
```

In Maude, we can use the up function to avoid the cumbersome task of explicitly writing the metarepresentation of a term or of a module. For example, to obtain the metarepresentation of the term s 0 in the module NAT, mathematically denoted $\overline{\texttt{s 0}}$, we can write

```
Maude> red up(NAT, s 0) .
Result Term : 's_['0.Nat]
```

Note that the module name is the first argument of the up function, with the term of that module to be metarepresented as the second argument. Since the same term can be parsed in different ways in different modules, and therefore can have different metarepresentations depending on the module in which it is considered, the module to which the term belongs has to be used to obtain the correct metarepresentation. Note also that the above reduction only makes sense at the metalevel, that is, in a module importing the module META-LEVEL. Moreover, by evaluating in any module importing the module META-LEVEL the up function with the name of any previously declared module as argument, we obtain the metarepresentation of such a module.

The result of a metalevel computation that may use several levels of reflection can be a term or module metarepresented one or more times, which may be hard to read. To display the output in a more readable form we can use the down command, which is in a sense inverse to up, since it gives us back the term from its metarepresentation. The down command takes two arguments. The first argument is the name of the module to which the term to be returned belongs. The metarepresentation of the desired output term should be the result of the command given as second argument. Thus, we can give the following command:

```
Maude> down NAT :
          red-in META-NAT : metaReduce(NAT, up(NAT, 0 + s 0)) .
Result Nat : s 0
```

The use of up and down can be iterated with as many levels of reflection as we wish.

6 Internal Strategies

As already explained, system modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating. Therefore, we need to have good ways of controlling the rewriting inference process—which in principle could not terminate or could go in many undesired directions—by means of adequate *strategies*. This need has been addressed in other languages; for example, the ELAN language provides a strategy language to guide the rewrites and allows user extensions for such a language [5,6,4]. In Maude, thanks to its reflective capabilities, strategies are made *internal* to the logic, that is, they are defined by rewrite rules in a normal module in Maude, and can be reasoned about as with rules in any other module.

In fact, there is great freedom for defining many different types of strategies, or even many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed particular strategy language. A general methodology for defining internal strategy languages for reflective logics is introduced in [12]. In general, strategies for controlling the application of the rules are defined by using `metaReduce`, `metaApply`, etc., as building blocks, which are then combined to obtain more complex strategies.

Let us illustrate some of the possibilities with some strategies controlling the execution of the rule labelled `switch` in the following module `SWITCH`. [9]

```
mod SWITCH is
    protecting ARRAY(Int) .
    vars I J X Y : MachineInt .
    var L :  List(Tuple(Int, Int)) .
    crl [switch] : (I, X) L (J, Y) => (I, Y) L (J, X)
       if (I, X) L (J, Y) : NeArray(Int) .
endm
```

The `switch` rule rewrites a term of sort `Array(Int)` in the module `ARRAY(Int)` to another term in which two of the elements in it have been interchanged. Note that the condition in the rule ensures that it is only applied to valid integer arrays, resulting in another valid integer array; however, this rule is different from the rule `sort` in Sections 3.2 and 4.5, because it does not check whether the elements are out of place or not.

[9] The reader should compare this module with the modules `INT-SORTING` in Section 3.2 and `SORTING(X :: TOSET)` in Section 4.5. In particular, the imported module `ARRAY(Int)` is obtained as an instantiation of the parameterized module `ARRAY(X)` in Section 4.5, and is equivalent to the module `INT-ARRAY` discussed in Section 2.3.

The system thus described is highly concurrent, because the `switch` rule may be applied concurrently to many different positions in an array. Moreover, this rule gives rise to nondeterministic and nonterminating computations, and therefore we need to control by means of strategies the way in which it is applied.

Let us begin by illustrating the use of `metaApply` for rewriting a term of sort `Array(Int)` by applying the rule `switch` once at the top of the term without any specific substitution (argument `none` representing the empty substitution) and using the first possible match (last argument `0`). The operation `getTerm` is the selector extracting the first component from either a pair of sort `ResultPair` or a triple of sort `ResultTriple` (see Section 5.5).

```
Maude> red getTerm(metaApply(SWITCH,
                    (1, 5)(2, 4)(3, 3), 'switch, none, 0)) .
result Term : (1, 3)(2, 4)(3, 5)
```

This simple application of the rule does not have much interest by itself, but shows how it can be used for building more interesting strategies. For example, in this case we see how an array can be rewritten in several different ways, even considering a single rule and rewriting only at the top of the term. The function `findAllRews` in the module `ALL-ONE-STEP-REWRITES` below finds all possible one-step rewrites of a term using a given rule. More precisely, `findAllRews(`M`,`T`,`L`)`, with M a term of sort `Module`, T a term of sort `Term` metarepresenting a term of a sort in the module metarepresented by M, and L the label of a rule in M, returns the set of terms resulting from the application of the rule L in all possible different ways on term T in M by using `metaXapply`.[10]

```
fmod SET(X :: TRIV) is
    sort Set(X) .
    subsort X.Elt < Set(X) .
    op mt : -> Set(X) [ctor] .
    op _&_ : Set(X) Set(X) -> Set(X) [ctor assoc comm id: mt] .
    var E : X.Elt .
    eq E & E = E .
endfm

view Term from TRIV to META-LEVEL is
    sort Elt to Term .
endv
```

[10] The constant `maxMachineInt` is the largest integer in a given Maude implementation. It is guaranteed (due to virtual memory/address space limitations) that in a nondistributed implementation of Maude a term of depth greater than `maxMachineInt` cannot be built without running out of swap space.

```
fmod ALL-ONE-STEP-REWRITES is
    protecting SET(Term) .

    op findAllRews : Module Term Qid -> Set(Term) .
    op findAllRewsAux : Module Term Qid MachineInt -> Set(Term) .

    var T : Term .         var M : Module .
    var L : Qid .          var N : MachineInt .

    eq findAllRews(M, T, L) = findAllRewsAux(M, T, L, 0) .

    eq findAllRewsAux(M, T, L, N)
      = if metaXapply(M, T, L, none, 0, maxMachineInt, N)
              :: Result4Tuple
        then getTerm(metaXapply(M, T, L, none, 0, maxMachineInt, N))
              & findAllRewsAux(M, T, L, N + 1)
        else mt
        fi .
endfm
```

A call to function `findAllRews` with the metarepresentations of the `SWITCH` module, of an array, and of the rule label `switch` gives back all the terms resulting from the application of such a rule in all possible ways on the term.

```
Maude> red findAllRews(SWITCH, (1, 5)(2, 4)(3, 3), switch) .
result Set(Term) :
    (1, 3)(2, 4)(3, 5) & (1, 4)(2, 5)(3, 3) & (1, 5)(2, 3)(3, 4)
```

It is easy to extend this specification in order to get not only the one-step rewrites, but also to get all rewrites, perhaps up to a given depth, and not only by the application of a single rule, but by considering any rule in a given module. We can even carry on some kind of model checking analysis. This is precisely the idea used by Denker, Meseguer, and Talcott in [21] for analyzing different communication protocols by means of exhaustive execution strategies that achieve a form of model checking analysis of the state space.

Another way of controlling the application of the rules consists in choosing some of the possible rewriting paths that can be followed by the application of the rules to a term. For example, we can consider different strategies for the controlled application of the rule `switch` above for sorting integer arrays. In this case, such strategies correspond to the specification of different *sorting algorithms* guiding where the `switch` rule should be applied at each point of the computation.

In the module `INSERT-STRATEGY` below, we give a strategy for sorting integer arrays by following the *insertion sort algorithm*. This strategy consists in partitioning the array in two regions: a first part which is sorted, and a second one which is unsorted. Initially, the entire array is unsorted, and, at each step, the strategy takes the first element of the unsorted part and places it into its correct position in the sorted region. This insertion requires the shifting of elements to make room for the element being inserted.

The function `insert` takes a term metarepresenting the nonempty array to be sorted, and calls an auxiliary function, named `insertAux`, which takes in addition the positions of the first and last elements of the array. Its second and third arguments are indexes used to refer to particular elements in the array. More precisely, the second argument represents the position of the first element in the unsorted region, that is, the element to be inserted next, and the third argument is used to go through the sorted region looking for the correct position for such an item.

The function `metaReduce` is used for reducing several expressions at the metalevel. For example, the term `T` being rewritten is used in `insert` for computing the range of the positions of the array, which are passed as arguments in the initial call to the `insertAux` function, or for evaluating the Boolean condition in which two elements in different positions are compared in order to decide whether it is worth to interchange them or not.

Note the form of the arguments of `metaReduce` in these calls. We use a combination of the overline notation with the actual metarepresentation of a term in order to simplify the text of the specification as much as possible. For example, the term `'_+_['low[T],` $\overline{1}$`]` is a simplified representation of the term `'_+_['low[T], '1.MachineInt]`, where `T` is a variable of sort `Term` with value the metarepresentation of an array. Such a term is the metarepresentation of `low(`A`) + 1`, with A the array metarepresented by `T`, which is used for calculating the successor of the first position of the array being sorted.

The function `metaXapply` is called with an explicit substitution as its fourth argument in order to appropriately instantiate the variables `I` and `J` used in the `switch` rule, corresponding to the positions whose values must be interchanged.

```
fmod INSERT-STRATEGY is
  protecting META-LEVEL(SWITCH) .

  op insert : Term -> Term .
  op insertAux : Term Term Term Term Term -> Term .

  vars T K1 K2 L H : Term .

  eq insert(T)
    = insertAux(T, '_+_['low[T], 1̅], '_+_['low[T], 1̅],
                'low[T], 'high[T]) .
```

```
  eq insertAux(T, K2, K1, L, H)
    = if getTerm(metaReduce(SWITCH, '_>_[K1, L])) == true
      then if getTerm(
                metaReduce(SWITCH,
                  '_>_['_`[_`][T, '_-_[K1, 1]], '_`[_`][T, K1]]))
              =/= true
           then insertAux(T, K2, '_-_[K1, 1], L, H)
           else insertAux(
                  getTerm(
                    metaXapply(SWITCH, T, 'switch,
                      (('J:MachineInt <-
                           getTerm(metaReduce(SWITCH, K1)));
                       ('I:MachineInt <-
                           getTerm(metaReduce(SWITCH, '_-_[K1, 1])))),
                      0, maxMachineInt, 0)),
                  K2, '_-_[K1, 1], L, H)
           fi
      else if getTerm(metaReduce(SWITCH, '_<_[K2, H])) == true
           then insertAux(T, '_+_[K2, 1], '_+_[K2, 1], L, H)
           else T
           fi
      fi .
endfm
```

Notice that, although the rule switch in module SWITCH gives rise to nondeterministic and nonterminating computations, its controlled application by means of the strategy insert in the module INSERT-STRATEGY is deterministic and terminating.

```
Maude> red insert((1,5)(2,4)(3,3)(4,2)(5,1)) .
result Term : (1,1)(2,2)(3,3)(4,4)(5,5)
```

We can specify other sorting algorithms following the same approach. For example, the module QUICKSORT-STRATEGY in Appendix A.2 defines the strategy function quicksort following the classical quicksort algorithm. Given a function partition, which partitions the array in those elements smaller than a chosen pivot and those greater than or equal to the pivot, the quicksort algorithm consists in calling partition with the fragment of the array being considered at that point, and then making recursive calls to itself with each of the fragments in which the selected pivot has divided the array.

7 Implementation

The Maude system is built around the Core Maude interpreter, which accepts module hierarchies of (unparameterized) functional and system modules with user-definable mixfix syntax. It is implemented in C++ and consists of two parts: the rewrite engine (Section 7.1) and the mixfix front end (Section 7.2). Two additional key components are the MSCP parser (Section 7.3) and the Full Maude language extension built on top of the Core Maude interpreter (Section 7.4).

7.1 The Rewrite Engine

The design of Maude's rewrite engine has a number of objectives. Specifically it should:

- look and feel like an interpreter;
- be capable of supporting user interrupts and source level tracing;
- be extensible with new equational theories and new built-in operators;
- be general purpose and not contain Maude-specific code or features.

The first three objectives all but rule out a number of performance enhancing techniques such as:

- compilation to native machine code (or C/C++);
- compilation to a fixed architecture abstract machine;
- program transformations and partial evaluation; and
- tight coupling between the matching/replacement/normalization code for different equational theories.

The design chosen is essentially a highly modular semicompiler where the most time consuming run-time tasks are compiled into a system of decision diagrams and automata which are interpreted at run time. It is realized as a C++ class library.

To enhance maintainability and extensibility, the rewrite engine is highly structured, with its classes being grouped into ten modules which themselves are organized into four layers, with inner layers having no knowledge of, or dependency on, classes in outer layers. The overall architecture is shown in Figure 2, where the arrows represent class derivation.

Layer 1 consists of a single module *Utility*, containing classes and class templates which provide a number of general-purpose data types such as vectors, maps, sets, graphs, and digraphs, together with some more specialized data

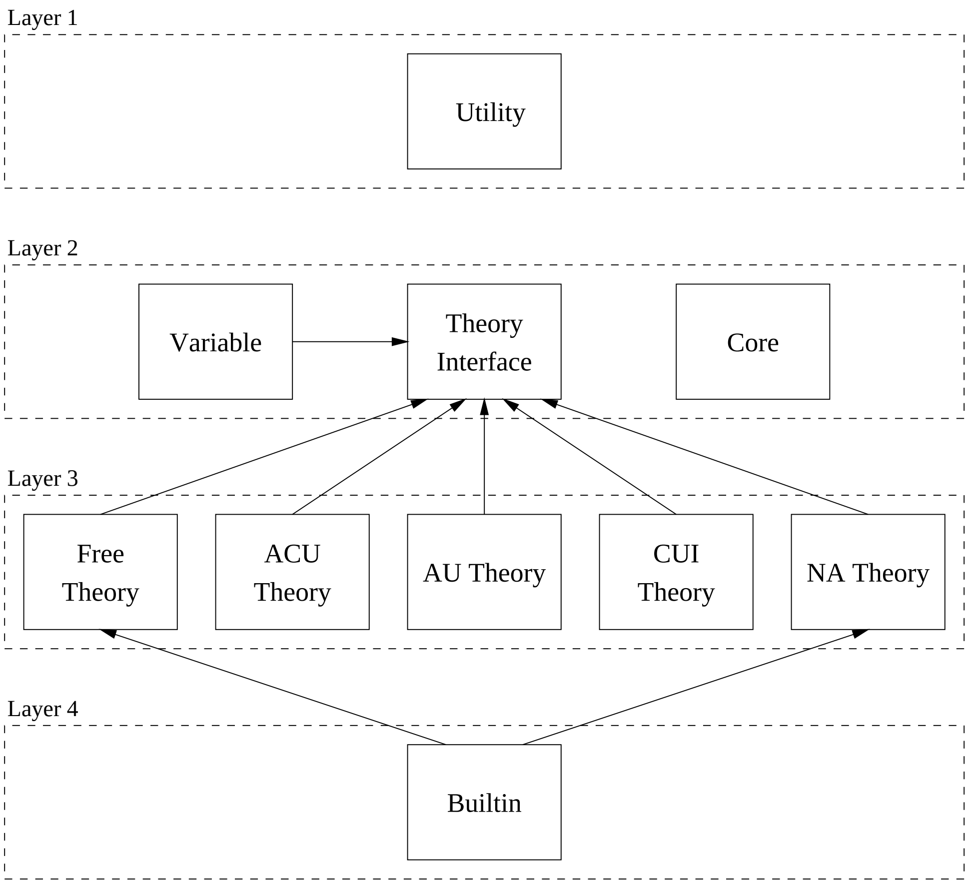

Fig. 2. Architecture of Maude's rewrite engine

types such as Diophantine equations and Tarjan's union-find data structure.

Layer 2 Consists of three modules. The *Theory Interface* module provides abstract interfaces to basic objects whose concrete realization will differ for different equational theories such as: symbols, term and DAG nodes, lefthand side automata (for matching), righthand side automata (for constructing and normalizing righthand side and condition instances), matching subproblems, and matching extension information. The *Core* module contains classes for basic objects that are independent of the different equational theories such as: sorts, connected sort components, equations, membership axioms, rules, conjunctions and disjunctions of matching subproblems, and substitutions. The *Variable* module contains classes derived from those in the Theory Interface. Variables are treated as a very special equational theory in that classes in most other modules are permitted to know about and depend on their special properties.

Layer 3 consists of modules that implement particular equational theories. Each consists of classes derived from those in the Theory Interface. Currently there are five such modules: The *Free Theory* module implements the free theory whose operators have no equational attributes; this is the only theory that currently supports many-to-one matching via discrimination nets. The *ACU Theory* module implements the associative-commutative and associative-commutative-identity theories. The *AU Theory* module implements the theories that contain associativity and possibly left and/or right identity. The *CUI Theory* module implements all theories that are formed by nonempty combinations of commutativity, left identity, right identity, and idempotence. The purpose of the *NA Theory* is to provide a convenient interface for plugging in data types such as machine integers, strings, and floating point numbers which have special machine level representations for performance reasons.

Layer 4 consists of a single module *Builtin* which contains classes for symbols with special built-in semantics, and for term and DAG nodes which have special internal representations. In keeping with our objective of having a general-purpose rewrite engine, this module provides operators and data types that are of general use in rewriting logic such as equality, sort tests, machine integers, strings, and floating point numbers.

7.1.1 Performance

Although our design emphasizes generality, transparency, extensibility, and maintainability, performance is not neglected. At the time of writing, typical equational rewriting speeds are 0.69-2.98M free-theory rewrites/second and 93-319K AC rewrites per second on a highend Linux PC (667MHz Xeon with 256MB 133MHz SDRAM). The figure for AC rewriting is highly dependent on the complexity of the AC patterns (AC matching is NP-complete) and the size of the AC subjects. These results were obtained using fairly simple linear and nonlinear patterns and large (hundreds of nested AC operators) subjects. In mixed free/AC systems we have obtained speeds of more than 1M rewrites/second.

Performance enhancing techniques used in the implementation include:

- Fixed size DAG nodes for in-place replacement.
- Full indexing for the topmost free function symbol layer of patterns; where the patterns for some free symbol only contain free symbols this is equivalent to matching a subject against all the patterns simultaneously.
- Use of *greedy matching algorithms* which attempt to generate a single matching substitution as fast as possible for patterns and subpatterns that are simple enough and whose variables satisfy certain requirements (such as not appearing in a condition). If a greedy matching algorithm fails it may be able to report that no match exists; but it is also allowed to report 'undecided', in which case the full matching algorithm must be used.
- Use of special-purpose matching automata to catch common subpatterns and handle them in a particularly efficient way.

- Use of a carefully-chosen normal form for the AC(U) theory, together with sophisticated renormalization algorithms that make use of extra information saved by the matcher to avoid costly comparisons and sorting where possible.
- Use of a Boyer-Moore style algorithm for matching under A(U) function symbols.
- Parse time analysis of sort information to avoid needless searching during A(U) and AC(U) matching.
- Parse time analysis of non-linear variables in patterns in order to propagate constraints on those variables in an 'optimal' way and reduce the search space.
- Global sort analysis to avoid unnecessary sort computations and tests.
- Compilation of sort information into ordered decision diagrams for fast incremental computation of sorts at run time.
- Efficient handling of *matching with extension* through a theory independent mechanism that avoids the need for extension variables or equations.

7.2 The Mixfix Frontend

The mixfix frontend contains all of the Maude-specific code in the system. It contains:

- A bison/flex-based parser for Maude's surface syntax.
- A grammar generator which generates the context-free grammar (CFG) for the mixfix parts of Maude over the user's signature.
- The MSCP parser for β-extended CFGs (discussed in Section 7.3 below).
- A mixfix pretty printer which is aware of precedences, gather patterns, and various kinds of overloading.
- A module system with lazy flattening and lazy reparsing (for when a module with dependents is replaced).
- A fully reentrant debugger.
- Maude-specific built-in data types, such as those in the `QID` and `META-LEVEL` modules.
- File, directory, and line number management.

7.3 The MSCP Parser

The intrinsic characteristics of Maude—mainly, its metalanguage functionality, its reflective nature, and its logical and semantic framework applications—pose very strong requirements on the design of a parsing algorithm for the language, since it has to fulfill the following constraints [60]:

- Interpreted parsing is required, since the syntax of modules is user-definable.
- Full context-free grammars must be used, and not only LALR models.
- A disambiguation mechanism, as the use of precedence values and gathering patterns, that modify the grammatical power of nonterminal symbols, must be available.
- Grammars are extended to incorporate *bubbles* [60]. Bubbles are the key notion to implement syntactic reflection. Furthermore, bubble sorts are user-definable.
- Techniques for error detection and error recovery must be supported.
- Efficiency is a main goal, as the parser is the surface of the rest of the system, especially in metalevel computations.

The logical kernel of the current version of the parser is based on the SCP parsing algorithm [59]. SCP is a bidirectional, bottom-up and event-driven parser for unrestricted context-free grammars. The soundness and completeness of SCP guarantees that the Maude version of SCP (MSCP) will generate all the possible grammatical analyses for each term in a given signature. This avoids some completeness problems detected in the OBJ3 parser.

MSCP is able to analyze *β-extended CFGs* (CFGs extended with bubbles and precedence/gathering patterns) [60]. The MSCP parsing algorithm incorporates sophisticated error detection and error recovery mechanisms based on the notions of partial derivability and adjacency, originally developed in SCP.

7.4 Full Maude

The full syntax of Maude explained and illustrated in this paper is not directly supported by the Core Maude interpreter. Instead, it is supported by a system extension called Full Maude [30,27] that is entirely written in Maude and makes crucial use of Maude's reflective capabilities. Specifically, all object-oriented features, as well as all parameterized modules, theories, views, and module expressions are supported in Full Maude. Essentially, Full Maude provides a rich and extensible *module algebra* of parameterized modules and module composition in the Clear/OBJ style with important extensions to support object-oriented modules. The key idea of its reflective design is to extend the sort `Module` in `META-LEVEL` with new sorts corresponding to more general kinds of modules and other constructs such as object-oriented modules, parameterized modules, theories, views, and so on. Then, all operations in the module algebra are defined by equations and rewrite rules at the metelevel.

As mentioned above, all of Full Maude—including its grammar, user interface, and internal functionality—has been formally specified in Maude using reflection. This formal specification is in fact its implementation. Our experience in this regard is very encouraging in several respects. Firstly, because of how quickly it was possible to develop Full Maude. Secondly, because of how easy it will be to maintain it, modify it, and extend it with new features and new module operations [28]. Thirdly, because of the competitive performance with which it can carry out complex module composition and module transformation operations, that makes the user interaction quite reasonable.

8 Methodology, Tools, Applications, and Future

We first explain how Maude, together with an environment of formal analysis and reasoning tools, can support a flexible range of formal methods. Then, after giving a brief summary of the different kinds of applications developed so far, we describe some near-future development concerning Mobile Maude.

8.1 Formal Methodology and Tools

The fact that rewriting logic specifications are executable allows us to have a flexible range of increasingly stronger formal methods, to which a system specification can be subjected, including the following:

(1) *Formal specification.* This process results in a first *formal model* of the system, in which many ambiguities and hidden assumptions present in an informal specification are clarified. A rewriting logic specification provides a formal model in exactly this sense.
(2) *Execution of the specification.* Executable rewriting logic specifications can be used directly for simulation and debugging purposes, leading to increasingly better designs.
(3) *Model-checking analysis.* Errors in highly distributed and nondeterministic systems not revealed by a particular execution can be found by a model-checking analysis that considers all behaviors of a system from an initial state, up to some level or condition.
(4) *Narrowing analysis.* By using symbolic expressions with logical variables, one can carry out a symbolic model-checking analysis in which all behaviors not only from a single initial state, but also from the possibly infinite set of states described by a symbolic expression are analyzed.
(5) *Formal proof.* For highly critical properties it is also possible to carry out a formal proof of correctness, which can be assisted by formal tools such as those described below. Such properties can be expressed in rewriting logic itself, or in an adequate modal or temporal logic.

The above methodology can be supported by formal tools. First of all, Maude itself is a very versatile formal tool supporting methods 1–2 through its default interpreter, and method 3 through reflective rewriting strategies that can analyze the different concurrent computations from a given initial state checking for desired properties. Method's 4 narrowing analysis can be supported by strategies and a rewriting specification of unification, but in the future it will be more efficient to support unification in a built-in way.

In addition to the formal methods directly supported by Maude, one can use Maude as a *formal metatool* [18] to build other formal tools supporting other kinds of analysis and proof. As explained in [17,18,51], reflection and the flexible uses of rewriting logic as a logical framework [41] are the key features making it easy to develop such formal tools and their user interfaces. The papers [18,57] give detailed accounts of a wide range of formal tools that have been defined in Maude by different authors for different formalisms. We focus here on Maude-specific tools, applicable to large classes of Maude specifications, or extensions of such specifications; they include the following:

An Inductive Theorem Prover. Using the reflective features of Maude, we have built an inductive theorem prover for equational logic specifications [17] that can be used to prove inductive properties of both CafeOBJ specifications [23] and of functional modules in Maude. This tool can be extended with reflective reasoning principles to reason about the metalogical properties of a logic represented in rewriting logic or, more generally, to prove metalevel properties [2].

A Church-Rosser Checker. We have also built a Church-Rosser checker tool [17] that analyzes equational specifications to check whether they satisfy the Church-Rosser property. This tool can be used to analyze order-sorted equational specifications [36] in CafeOBJ and in Maude. The tool outputs a collection of proof obligations that can be used to either modify the specification or to prove them. Extensions of this tool to perform equational completion and to check coherence of rewrite theories are currently under development.

Real-Time Maude. Based on a notion of *real-time rewrite theory* that can naturally represent many existing models of real-time and hybrid systems, and that has a straightforward translation into an ordinary rewrite theory [58,56], Ölveczky and Meseguer have developed an execution and analysis environment for specifications of real-time and hybrid systems called Real-Time Maude [57]. This tool translates real-time rewrite theories into Maude modules and can execute and analyze such theories by means of a library of strategies that can be easily extended by the user to perform other kinds of formal analysis.

8.2 Applications

In general, the applications of Maude exploit the good features of rewriting logic as a semantic framework and as a logical framework. Often, they use in a crucial way Maude's reflective capabilities. A detailed discussion of different applications is beyond the scope of this paper; we refer the reader to [47,49,18,21,51,50] for recent accounts. As already explained in Section 8.1, an important class of logical framework applications are formal metatool applications that use Maude to generate other formal tools [18]. Semantic framework applications span a wide range of levels, including: formal specification of architectural description languages, object-oriented designs, and distributed middleware [49,50]; formal specification and analysis of network systems and communication protocols [21,50]; and specification and programming of agent and mobile systems (see [50,29] and Section 8.3). Of course, given the high performance of the implementation, Maude is also an attractive very high-level language for a number of programming applications. As explained below, we expect Mobile Maude to further extend the range of such applications.

8.3 Mobile Maude

Maude can be used not only for specifying communication systems, but also for programming them. We are currently advancing the design of Mobile Maude [29]. This is an extension of Maude supporting mobile computation that uses reflection in a systematic way to obtain a simple and general declarative mobile language design. The two key notions are *processes* and *mobile objects*. Processes are located computational environments where mobile objects can reside. Mobile objects can move between different process in different locations, and can communicate asynchronously with each other by means of messages. Each mobile object contains its own code—that is a rewrite theory $\mathcal{R}$—metarepresented as a term $\overline{\mathcal{R}}$. In this way, reflection endows mobile objects with powerful "higher-order" capabilities within a simple first-order framework.

We expect that Mobile Maude will have good support for *secure* mobile computation for two reasons. Firstly, mobile objects will communicate with each other and will move from one location to another using state-of-the-art encryption mechanisms. Secondly, because of the logical basis of Mobile Maude, we expect to be able to prove critical properties of applications developed in it with much less effort than what it would be required if the same applications were developed in a conventional language such as Java.

9 Maude Versions: Past, Present, and Future

As explained in the introduction, this paper has presented all the main Maude concepts in a version-independent way, without pointing out for each language feature in which version it was introduced. Table 1 summarizes this information, and also distinguishes at the same time between the Core Maude features, and the additional features provided in Full Maude.

Version 1 of Maude was released in January 1999, while Version 2 was designed in the summer of 2000; most of its features are already implemented at the time of writing. The last row in the table summarizes several features that have been discussed as desirable for future versions, but that are not going to be part of the release of Version 2 of Maude.

A More details of some examples

A.1 CCS Syntax

```
fmod ACTION is
    protecting QID .
    sorts Label Act .
    subsorts Qid < Label < Act .
    op tau : -> Act [ctor] .  *** silent action
    op ~_ : Label -> Label [ctor] .
    var N : Label .
    eq ~ ~ N = N .
endfm

fmod PROCESS is
    protecting ACTION .
    sorts ProcessId Process .
    subsorts Qid < ProcessId < Process .
    op 0 : -> Process [ctor] .
         *** inaction
    op _._ : Act Process -> Process [ctor] .
         *** prefix
    op _+_ : Process Process -> Process [ctor assoc comm] .
         *** summation
    op _|_ : Process Process -> Process [ctor assoc comm] .
         *** parallel composition
    op _\_ : Process Label -> Process [ctor] .
         *** restriction
    op _[_/_] : Process Label Label -> Process [ctor] .
```

<table>
<tr><th></th><th>Core Maude</th><th>Full Maude</th></tr>
<tr><td>Version 1</td><td>functional modules
system modules
conditions: single equation
module hierarchies
reflection (metalevel)
internal strategies
descent functions { metaReduce, metaRewrite, metaApply }
predefined data types { Boolean values, quoted identifiers, machine integers }</td><td>object-oriented modules
parameterized modules
theories
views
module renaming
tuples
up/down commands</td></tr>
<tr><td>Version 2</td><td>explicit use of kinds
new variable syntax
general conditions { memberships, equations, matching equations, rewrites }
more descent functions { metaXapply, metaMatch, metaXmatch }
more predefined data types { natural numbers, floating point numbers, strings }
built-in object-oriented modules, including TCP socket and file system interfaces
fair rewriting for system and object-oriented modules
rewrite search and LTL model-checking
sublanguage compiler
LaTeX pretty printing</td><td>parameterized theories
parameterized views
view composition
view lifting</td></tr>
<tr><td>Future</td><td>unification
narrowing
built-in strategy language
foreign language interface
user-definable lexical syntax
GUI support
additional operator attributes
additional compiler support</td><td></td></tr>
</table>

Table 1
Language Features

```
        *** relabelling: [b/a] relabels "a" to "b"
endfm

fmod CCS-CONTEXT is
    protecting PROCESS .
    sort Context .
    op _=def_ : ProcessId Process -> Context [ctor] .
    op nil : -> Context [ctor] .
    op _&_ : Context Context -> [Context]
         [ctor assoc comm id: nil] .
    op _definedIn_ : ProcessId Context -> Bool .
    op def : ProcessId Context -> [Process] .
    op not-defined : -> [Process] [ctor] .
    op context : -> Context .
    vars X X' : ProcessId .   var P : Process .
    vars C C' : Context .

    cmb (X =def P) & C : Context if not(X definedIn C) .
    eq X definedIn nil = false .
    ceq X definedIn C = (X == X') or (X definedIn C')
         if (X' =def P) & C' := C .
    eq def(X, nil) = not-defined .
    ceq def(X, C) = P if (X =def P) & C' := C .
    ceq def(X, C) = def(X, C')
         if (X' =def P) & C' := C /\ X =/= X' .
endfm
```

A.2 Quicksort Strategy

The following module `QUICKSORT-STRATEGY` defines the `quicksort` strategy function, which follows the classical quicksort algorithm for sorting. There is an auxiliary function `quicksortAux` taking two additional arguments, namely the positions of the first and last elements to be considered by the function, that is, the limits of the fragment being considered in each call. There is another auxiliary function `partition`, which takes as pivot the first of the elements in the fragment of the array being considered, and returns a pair of terms (of sort `Tuple(Term, Term)`) which metarepresent, respectively, the resulting array and the position of the pivot element in it, in such a way that all the elements before such a position are smaller than the pivot, and all the elements after it are greater than or equal to the pivot. The position of the pivot in the resulting array is used by the function `quicksortAux` for making the recursive calls. Thus, given a fragment with first position L and last position H, and with P the position of the pivot after the call to `partition`, the recursive calls will be made with fragments $L, P-1$ and $P+1, H$. Note that the module expression TUPLE[2](Term, Term) provides a sort Tuple(Term, Term) with constructor (_,_), and with projection functions p1_ and p2_.

```
fmod QUICKSORT-STRATEGY is
    protecting META-LEVEL(SWITCH) + TUPLE[2](Term, Term) .

    op quicksort : Term -> Term .
    op quicksortAux : Term Term Term -> Term .
    op partition : Term Term Term Term -> Tuple(Term, Term) .

    vars T P L H : Term .

    eq quicksort(T) = quicksortAux(T, 'low[T], 'high[T]) .

    eq quicksortAux(T, L, H)
      = if getTerm(metaReduce(SWITCH, '_>_[L, H])) == true
        then nil
        else if getTerm(metaReduce(SWITCH, '_==_[L, H])) == true
             then '`(_`,_`)[
                           getTerm(metaReduce(SWITCH, '_`[_`][T, L])), L]
             else '__[quicksortAux(p1 partition(T, L, '_+_[L, 1], H),
                        L,'_-_[p2 partition(T, L, '_+_[L, 1], H), 1]),
                      '`(_`,_`)[p2 partition(T, L, '_+_[L, 1], H),
                        '_`[_`][p1 partition(T, L, '_+_[L, 1], H),
                                p2 partition(T, L, '_+_[L, 1], H)]],
                      quicksortAux(
                        p1 partition(T, L, '_+_[L, 1], H),
                        '_+_[p2 partition(T, L, '_+_[L, 1], H), 1], H)]]
             fi
        fi .

    eq partition(T, P, L, H)
      = if getTerm(metaReduce(SWITCH, '_>_[L, H])) == true
        then if getTerm(metaReduce(SWITCH, '_<_[P, H])) == true
             then (getTerm(          *** move the pivot to position H
                     metaXapply(SWITCH, T, 'switch,
                       (('I:MachineInt <-
                            getTerm(metaReduce(SWITCH, P)));
                        ('J:MachineInt <-
                            getTerm(metaReduce(SWITCH, H)))),
                       0, maxMachineInt, 0)), H)
             else (T, P)          *** The pivot is the biggest element
             fi
        else if getTerm(
                  metaReduce(SWITCH, '_>_['_`[_`][T, P],'_`[_`][T, L]]))
                == true
             then      *** the element at L is smaller than the pivot
```

```
              partition(T, P, '_+_[L, 1], H)
          else if getTerm(
                    metaReduce(SWITCH,
                      '_<=_['_`[_`][T, P], '_`[_`][T, H]]))
                  == true
               then *** the element at H is greater than the pivot
                    partition(T, P, L, '_-_[H, 1])
               else partition(
                      getTerm(
                        metaXapply(SWITCH, T, 'switch,
                          (('I:MachineInt <-
                               getTerm(metaReduce(SWITCH, L)));
                           ('J:MachineInt <-
                               getTerm(metaReduce(SWITCH, H)))),
                          0, maxMachineInt, 0)),
                      P, '_+_[L, 1], '_-_[H, 1])
               fi
          fi
     fi .
endfm
```


Acknowledgements

We would like to thank David de Frutos, Miguel Palomino, Alberto Verdejo, and the anonymous referees for all their helpful comments to previous versions of this paper.

References

[1] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports. Springer-Verlag, 1999.

[2] D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. In S. Kapoor and S. Prasad, editors, *Proceedings Twentieth Conference on the Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 13–15*, volume 1974 of *Lecture Notes in Computer Science*, pages 55–80. Springer-Verlag, 2000.

[3] J. Bergstra and J. Tucker. 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, The Netherlands*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.

[4] P. Borovanský. *Le Contrôle de la Réécriture: Étude et Implantation d'un Formalisme de Stratégies.* PhD thesis, Université Henri Poincaré – Nancy I, Oct. 1998.

[5] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In Meseguer [46], pages 168–188. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[6] P. Borovanský, C. Kirchner, and H. Kirchner. Rewriting as a unified specification tool for logic and control: The ELAN language. In M. P. A. Sellink, editor, *Second International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, The Netherlands, September 25–26, 1997*, Electronic Workshops in Computing. Springer-Verlag, 1998. `http://www.ewic.org.uk/ewic/workshop/view.cfm/ASFSDF-97`.

[7] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In Kirchner and Kirchner [39], pages 329–344. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[8] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In Meseguer [46], pages 35–50. `http://www.elsevier.nl/locate/entcs/volume4.html`.

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

[10] R. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In D. Bjørner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer-Verlag, 1980.

[11] G. Carabetta, P. Degano, and F. Gadducci. CCS semantics via proved transition systems and rewriting logic. In Kirchner and Kirchner [39], pages 253–272. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[12] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, 2000.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In Kirchner and Kirchner [39], pages 3–24. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. Manual distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. `http://maude.csl.sri.com/manual`, Jan. 1999.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. A Maude tutorial. Tutorial distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. Presented at the *European Joint Conference on Theory and Practice of Software,*

ETAPS 2000, Berlin, Germany, March 25, 2000. http://maude.csl.sri.com/tutorial, Mar. 2000.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In Futatsugi [34], pages 297–318. http://www.elsevier.nl/locate/entcs/volume36.html.

[17] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, Apr. 1998. http://maude.csl.sri.com/papers.

[18] 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.

[19] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [46], pages 65–89. http://www.elsevier.nl/locate/entcs/volume4.html.

[20] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 2001. This volume.

[21] G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In D. Maughan, G. Koob, and S. Saydjari, editors, *Proceedings DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head Island, South Carolina, January 25–27, 2000*, pages 251–265. IEEE Computer Society Press, 2000. http://schafercorp-ballston.com/discex/.

[22] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*, chapter 6, pages 243–320. The MIT Press/Elsevier, 1990.

[23] R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[24] R. Diaconescu, K. Futatsugi, and S. Iida. Component-based algebraic specification and verification in CafeOBJ. 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 1644–1663. Springer-Verlag, 1999.

[25] R. Diaconescu, K. Futatsugi, M. Ishisone, T. Sawada, and A. T. Nakagawa. An overview of CafeOBJ. In Kirchner and Kirchner [39], pages 75–88. http://www.elsevier.nl/locate/entcs/volume15.html.

[26] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularization. In G. Huet, G. Plotkin, and C. Jones, editors, *Proceedings of Workshop on Logical Frameworks (Edinburgh, United Kingdom, May 1991)*, pages 83–130. Cambridge University Press, May 1991.

[27] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Spain, June 1999. `http://maude.csl.sri.com/papers`.

[28] F. Durán. The extensibility of Maude's module algebra. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 422–437. Springer-Verlag, 2000.

[29] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13–15, 2000, Proceedings*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2000.

[30] F. Durán and J. Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [39], pages 185–206. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[31] F. Durán and J. Meseguer. Parameterized theories and views in Full Maude 2.0. In Futatsugi [34], pages 319–337. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[32] S. Eker. Fast matching in combination of regular equational theories. In Meseguer [46], pages 90–108. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[33] S. Eker. Term rewriting with operator evaluation strategy. In Kirchner and Kirchner [39], pages 45–62. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[34] K. Futatsugi, editor. *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[35] F. Gadducci and U. Montanari. Comparing logics for rewriting: Rewriting logic, action calculi and tile logic. *Theoretical Computer Science*, 2001. This volume.

[36] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[37] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software*

Engineering with OBJ: Algebraic Specification in Action, Advances in Formal Methods, chapter 1, pages 3–167. Kluwer Academic Publishers, 2000.

[38] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15:1155–1194, Nov. 1986.

[39] 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*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[40] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentenryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. The MIT Press, 1995.

[41] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, Aug. 1993. To appear in D. Gabbay, editor, *Handbook of Philosophical Logic, Second Edition, Volume 6*, Kluwer Academic Publishers, 2001. `http://maude.csl.sri.com/papers`.

[42] J. Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, Feb. 1990. Revised June 1990.

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

[44] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

[45] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. M. Nierstrasz, editor, *ECOOP'93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26–30, 1993, Proceedings*, volume 707 of *Lecture Notes in Computer Science*, pages 220–246. Springer-Verlag, 1993.

[46] J. Meseguer, editor. *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Sept. 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[47] J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory, 7th International Conference, Pisa, Italy, August 26–29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer-Verlag, 1996.

[48] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[49] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, Proceedings of the NATO Advanced Study Institute on Computational Logic held in Marktoberdorf, Germany, July 29 – August 6, 1997*, volume 165 of *NATO ASI Series F: Computer and Systems Sciences*, pages 347–398. Springer-Verlag, 1998.

[50] J. Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In S. F. Smith and C. L. Talcott, editors, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems IV, FMOODS 2000, September 6–8, 2000, Stanford, California, USA*, pages 89–117. Kluwer Academic Publishers, 2000.

[51] J. Meseguer. Rewriting logic and Maude: Concepts and applications. In L. Bachmair, editor, *Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10–12, 2000, Proceedings*, volume 1833 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2000.

[52] J. Meseguer and J. A. Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 104(1):114–158, 1993.

[53] J. Meseguer and C. L. Talcott. A partial order event model for concurrent objects. In J. C. M. Baeten and S. Mauw, editors, *CONCUR'99, Concurrency Theory, 10th International Conference Eindhoven, The Netherlands, August 24–27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 415–430. Springer-Verlag, 1999.

[54] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[55] T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, 12:633–653, 1991.

[56] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic.* PhD thesis, University of Bergen, Norway, 2000. `http://maude.csl.sri.com/papers`.

[57] P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In Futatsugi [34], pages 361–383. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[58] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 2001. This volume. `http://maude.csl.sri.com/papers`.

[59] J. F. Quesada. *The SCP parsing algorithm based on syntactic constraints propagation.* PhD thesis, Universidad de Sevilla, Spain, June 1997.

[60] J. F. Quesada. The Maude parser: Parsing and meta-parsing β-extended context-free grammars. Technical report, Computer Science Laboratory, SRI International, 2001. To appear.

[61] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In J. Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA'95, Kaiserslautern, Germany, April 5–7, 1995, Proceedings*, volume 914 of *Lecture Notes in Computer Science*, pages 179–193. Springer-Verlag, 1995.

[62] C. L. Talcott. An actor rewriting theory. In Meseguer [46], pages 360–383. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[63] C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J.-B. Stefani, editors, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS'96*, pages 154–169. Chapman & Hall, 1997.

[64] C. L. Talcott. Towards a toolkit for actor system specification. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 391–406. Springer-Verlag, 2000.

[65] C. L. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 2001. This volume.

[66] A. Verdejo and N. Martí-Oliet. Executing and verifying CCS in Maude. Technical Report 99-00, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Feb. 2000. `http://maude.csl.sri.com/casestudies/ccs`.

[67] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4–8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer-Verlag, 1994.

[68] P. Viry. Rewriting modulo a rewrite system. Technical Report TR-95-20, Dipartimento di Informatica, Università di Pisa, Dec. 1995. `ftp://ftp.di.unipi.it/pub/techreports/TR-95-20.ps.Z`.