Electronic Notes in Theoretical Computer Science 4 (1996)

Principles of Maude

M. Clavel [3], S. Eker [1,3], P. Lincoln [2,3], and J. Meseguer [3]

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA

Abstract

This paper introduces the basic concepts of the rewriting logic language Maude and discusses its implementation. Maude is a wide-spectrum language supporting formal specification, rapid prototyping, and parallel programming. Maude's rewriting logic paradigm includes the functional and object-oriented paradigms as sublanguages. The fact that rewriting logic is reflective leads to novel metaprogramming capabilities that can greatly increase software reusability and adaptability. Control of the rewriting computation is achieved through internal strategy languages defined inside the logic. Maude's rewrite engine is designed with the explicit goal of being highly extensible and of supporting rapid prototyping and formal methods applications, but its semi-compilation techniques allow it to meet those goals with good performance.

1 Introduction

Maude is a logical language based on rewriting logic [17,24,20]. It is therefore related to other rewriting logic languages such as Cafe [10], ELAN [12], and DLO [6]. The equational language OBJ [11] can be regarded as a functional sublanguage of Maude.

This paper gives an introduction to the language and its interpreter implementation. Particular emphasis is placed on its basic principles and on its semantics. The style is informal, and the ideas are illustrated with simple examples to facilitate their comprehension.

The key characteristics of Maude can be summarized as follows:

[1] Supported by a NATO Fellowship administered through the Royal Society.

[2] Supported in part by Office of Naval Research Contract N00014-95-C-0168, National Science Foundation Grant CCR-9224858, and AFOSR Contract number F49620-95-C0044.

[3] Supported by Office of Naval Research Contracts N00014-95-C-022, and N00014-96-C-0114, National Science Foundation Grant CCR-9224005, DARPA through NRAD Contract N66001-95-C-8620, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

© *1996 Elsevier Science B. V.*

- *Based on rewriting logic.* This makes it particularly well suited to express in a declarative way concurrent and state-changing aspects of systems. Programs are theories, and rewriting logic deduction exactly corresponds to concurrent computation.
- *Wide-spectrum.* Rewriting logic is a logical and semantic framework in which specification, rapid prototyping, and efficient parallel and distributed execution, as well as formal transformations from specifications to programs can be naturally supported [14].
- *Multiparadigm.* Since rewriting logic conservatively extends equational logic [15], a equational style of functional programming is naturally supported in a sublanguage. A declarative style of concurrent object-oriented programming is also supported with a simple logical semantics. Since rewriting logic also extends Horn logic with equality in a conservative way [15], Horn logic programming can also be supported and extended in an implementation with basic facilities for unification.
- *Reflective.* Rewriting logic is reflective [8,7]. The design of Maude capitalizes on this fact to support a novel style of *metaprogramming* with very powerful module-combining and module-transforming operations that surpass those of traditional parameterized programming and can greatly advance software reusability and adaptability.
- *Internal Strategies.* The strategies controlling the rewriting process can be defined by rewrite rules and can be reasoned about inside the logic. Therefore, instead of having a "Logic+Control" introduction of extra-logical features, in Maude "Control $\subseteq$ Logic."

Maude's implementation has been designed with the explicit goals of supporting executable specification and formal methods applications, of being easily extensible, and of supporting reflective computations. Although it is an interpreter, its advanced semi-compilation techniques support flexibility and traceability without sacrificing performance. It can reach up to 200,000 rewrites per second on some applications running on a 90 MHz Sun HyperSPARC.

Section 2 explains the sublanguage of functional modules. An informal introduction to rewriting logic and to object-oriented modules in given in Section 3. System modules, reflection, and internal strategies are discussed in Section 4. Maude's metaprogramming capabilities are the subject of Section 5. Section 6 summarizes the semantic foundations of the language, and Section 7 describes the interpreter implementation. We conclude with some plans for the future.

2 Functional Modules

Functional modules define data types and functions on them by means of equational theories whose equations are Church-Rosser and terminating. A mathematical model of the data and the functions is provided by the *initial algebra* defined by the theory, whose elements consist of equivalence classes

of ground terms modulo the equations. Evaluation of any expression to its reduced form using the equations as rewrite rules assigns to each equivalence class a unique canonical representative. Therefore, in a more concrete way we can equivalently think of the initial algebra as consisting of those canonical representatives; that is, of the values to which the functional expressions evaluate.

As in the OBJ language [11] that Maude extends, functional modules can be unparameterized, or they can be parameterized with *functional theories* as their parameters. Functional theories have a "loose semantics," as opposed to an initial one, in the sense that any algebra satisfying the equations in the theory is an acceptable model. For example, a parameterized list module `LIST[X :: TRIV]` forms lists of models of the trivial parameter theory

```
fth TRIV is
  sort Elt .
efth
```

with one sort `Elt`; those models as just sets of elements. Similarly, a sorting module `SORTING[Y :: POSET]` sorts lists whose elements belong to a model of the `POSET` functional theory, that is, the elements must have a partial order.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called *membership equational logic* [16,3]; we discuss this and give more details about the semantics of functional modules in Section 6.1. For the moment, it suffices to say that, in addition to supporting sorts, subsorts, and overloading of function symbols, functional modules also support *membership axioms*, a generalization of sort constraints [23] in which a term is asserted to have a certain sort if a condition consisting of a conjunction of equations and of unconditional membership tests is satisfied.

We can illustrate these ideas with a parameterized module `PATH[G :: GRAPH]` that forms paths over a graph. This module has a path concatenation operation, has nodes as identities, and source and target functions.

```
th GRAPH is
  sorts Node Edge .
  ops s t : Edge -> Node .  *** source and target
eth

fmod PATH[G :: GRAPH] is
  sorts Path Path? .
  subsorts Node Edge < Path < Path? .
  ops s t : Path -> Node .
  op _;_ : Path? Path? -> Path? .
  var E : Edge .
  var N : Node .
  var P : Path .
  vars Q R S : Path? .
  eq (Q ; R) ; S = Q ; (R ; S) .
  cmb E ; P : Path if t(E) == s(P) .
```

```
  eq s(N) = N .
  eq t(N) = N .
  ceq s(E ; P) = s(E) if t(E) == s(P) .
  ceq t(E ; P) = t(P) if t(E) == s(P) .
  ceq N ; P = P if s(P) == N .
  ceq P ; N = P if t(P) == N .
endfm
```

Note that the concatenation of two paths is a path if and only if the target of the first is the source of the second. This follows as an inductive consequence of the simpler conditional membership axiom

```
  cmb E ; P : Path if t(E) == s(P) .
```

where `E` is an edge and `P` a path. We can then instantiate this module with a concrete graph corresponding to an automaton, and can evaluate path expressions to check whether they are valid paths in the automaton.

```
fmod AUTOMATON is
  sorts Node Edge .
  ops a b c : -> Node .
  ops f g h i j : -> Edge .
  ops s t : Edge -> Node .
  eq s(f) = a .  eq t(f) = b .
  eq s(g) = c .  eq t(g) = a .
  eq s(h) = b .  eq t(h) = c .
  eq s(i) = c .  eq t(i) = b .
  eq s(j) = b .  eq t(j) = b .
endfm

make RECOGNIZER is PATH[AUTOMATON] endm
```

3 Rewriting Logic and Object-Oriented 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 by further rewriting.

The essential idea of rewriting logic [19] 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. Of course, rewriting will happen *modulo* whatever structural axioms the state of the system satisfies. For example, the top level of a distributed system's state does often have the structure of a *multiset*, so that we can regard the system as composed together by an associative and commutative state constructor.

We can represent a *rewrite theory* as a four-tuple $\mathcal{R} = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic, that specifies states of the system as an abstract data type, L is a set of labels, to label the rules, and R is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in R may be conditional [19].

Rewriting logic is therefore a logic of concurrent state change. The logic's four rules of deduction—namely, reflexivity, transitivity, congruence, and replacement [19]—allow us to infer all the complex concurrent state changes that a system may exhibit, given a set of rewrite rules that describe its elementary local changes. It then becomes natural to realize that many reactive systems so specified should never terminate, and that a system may evolve in highly nondeterministic ways through paths that will never cross each other.

These ideas can be illustrated by explaining how concurrent object-oriented systems can be specified in rewriting logic, and how they can be executed using Maude's object-oriented modules.

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. Therefore, we can view configurations as built up by a binary multiset union operator which we can represent with empty syntax as

```
subsorts Object Msg < Configuration .
op __ : Configuration Configuration -> Configuration
                                  [assoc comm idr: null] .
```

where the multiset union operator `__` is declared to satisfy the structural laws of associativity and commutativity and to have identity *null*. The subsort declaration

```
subsorts Object Msg < Configuration .
```

states that objects and messages are singleton multiset configurations, so that more complex configurations are generated out of them by multiset union.

As a consequence, we can abstractly represent the configuration of a typical concurrent object-oriented system as an equivalence class $[t]$ modulo the structural laws of associativity and commutativity obeyed by the multiset union operator of a term expressing a union of objects and messages, i.e., as a multiset of objects and messages.

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

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*.

The set of all the attribute-value pairs of an object state is formed by repeated application of the binary union operator _,_ which also obeys structural laws of associativity and commutativity; i.e., the order of the attribute-value pairs of an object is immaterial.

Consider for example a concurrent system made up of sender and receiver objects that communicate with each other by sending messages in an unreliable environment in which messages may be received out of order, some messages can be lost, and other messages can be duplicated. A fault-tolerant connection between two such objects can be accomplished by numbering the messages and sending acknowledgments back. A receiver object may have the form

```
< R : Receiver | from: S, recq: Q, reccnt: M >
```

where the attribute `from` is the name of the sending object, `recq` is the queue of received messages, and `reccnt` is the receiver's counter. In Maude, the class `Receiver` of such objects is specified by the declaration

```
class Receiver | from: OId, recq: Queue, reccnt: Nat .
```

that introduces the attribute names and the corresponding value sorts. The concurrent local state change corresponding to the reception of one message from the sender by the receiver object can then be described by the following labeled rewrite rule.

```
rl [ receive ] :
   < R : Receiver | from: S, recq: Q, reccnt: M >
   (to: R (E,N))
=> < R : Receiver | from: S,
                    recq: (if N == s(M) then push(Q,E) else Q fi),
                    reccnt: (if N == s(M) then s(M) else M fi) >
   (to: S ack N) .
```

That is, the new value `E` is appended to the queue and the counter is increased iff the number `N` in the message is `M + 1`; otherwise, the message is discarded and the receiver does not change its state, but in any case an acknowledgment is always sent to the sender.

The entire fault-tolerant protocol for sender and receiver objects—discussed in a somewhat different way in Chandy and Misra [5], and similar in some ways to the presentation of the alternating bit protocol by Lam and Shankar [13]—can be defined in the following parameterized object-oriented module.

Note that Maude's syntax for object-oriented modules leaves implicit some well-understood assumptions, such as the syntax for objects, the existence of a multiset union operator to form configurations, and the conventions for class inheritance. However, object-oriented modules can be systematically translated into ordinary rewrite theories by making explicit all these assumptions. They can therefore be understood as a special case of system modules. A detailed account of this translation process can be found in [20].

```
omod PROTOCOL[ELT :: TRIV] is
  protecting QUEUE[ELT] .
```

```
  sort Contents Count .
  subsort Elt < Contents .
  op z : -> Count .
  op s_ : Count -> Count .
  op empty : -> Contents .
  msg to:_(_,_) : OId Elt Count -> Msg . *** data to receiver
  msg to:_ack_ : OId Count -> Msg . *** acknowledgment to sender
  class Sender | rec: OId, sendq: Queue, sendbuff: Contents,
                 sendcnt: Count, repcount: Count .
  class Receiver | from: OId, recq: Queue, reccnt: Count .
  vars S R : OId .
  vars N M X : Count .
  var E : Elt .
  var Q : Queue .
  var C : Contents .

  rl [ produce ] :
     < S : Sender | rec: R, sendq: cons(E, Q), sendbuff: empty,
                    sendcnt: N, repcount: X > =>
     < S : Sender | rec: R, sendq: Q, sendbuff: E,
                    sendcnt: s(N), repcount: s(s(s(z))) > .

  rl [ send ] :
     < S : Sender | rec: R, sendq: Q, sendbuff: E,
                    sendcnt: N, repcount: s(X) > =>
     < S : Sender | rec: R, sendq: Q, sendbuff: E,
                    sendcnt: N, repcount: X >
     (to: R (E,N)) .

  rl [ rec-ack ] :
     < S : Sender | rec: R, sendq: Q, sendbuff: C,
                    sendcnt: N, repcount: X >
     (to: S ack M) =>
     < S : Sender | rec: R, sendq: Q,
                    sendbuff: (if N == M then empty else C fi),
                    sendcnt: N, repcount: X > .

  rl [ receive ] :
     < R : Receiver | from: S, recq: Q, reccnt: M >
     (to: R (E,N))
  => < R : Receiver | from: S,
                      recq: (if N == s(M) then push(Q,E) else Q fi),
                      reccnt: (if N == s(M) then s(M) else M fi) >
     (to: S ack N) .
endom
```

These definitions will generate a reliable, in-order communication mechanism

from an unreliable one. The message counts are used to ignore all out-of-order messages, and the replication count is used to replicate messages that may be lost if the channel is faulty. The fairness assumptions of Maude will ensure that the `send` action and corresponding `receive` actions will be repeated until a `rec-ack` can be performed, or the replication counter goes to zero. One can directly represent unbounded retransmission by eliminating this check as well, although the protcol then relies more strongly on fairness assumption. In [24,20] it is explained how we can also model some fault modes of the communication channel by additional rewrite rules which duplicate or destroy messages declared in a module extending the one above.

Formally, letting C denote the initial configuration of objects and C' denote configuration resulting after rewriting, we have been able to deduce the sentence $C \longrightarrow C'$ as a logical consequence of the rewrite rules in the module. Indeed, the rules of deduction of rewriting logic support sound and complete reasoning about the concurrent transitions that are possible in a concurrent system whose basic local transitions are axiomatized by given rewrite rules. That is, the sentence $[t] \longrightarrow [t']$ is provable in the logic using the rewrite rules that axiomatize the system as axioms if and only if the concurrent transition $[t] \longrightarrow [t']$ is possible in the system.

In this object-oriented case we make several implicit assumptions, including the associativity and commutativity of the multiset union operator. In general system modules, however, the axioms E can be varied as a very flexible parameter to specify many different types of concurrent systems. In this way, rewriting logic can be regarded as a very general semantic framework for concurrency that encompasses a very wide range of well-known models [19,22].

Maude's default interpreter can be quite adequate for simulating concurrent object-oriented systems. However, for the purposes of studying a system in depth—for example, by exploring all the possible rewrites from a given state to another—or of controlling the possibly highly nondeterministic evolution of a system that need not be object-oriented, we need other means.

4 System Modules, Strategies, and Reflection

The most general Maude modules are system modules. They specify the initial model of a rewrite theory $\mathcal{R}$ [19]. This initial model is a transition system 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 computations in the system so described. Such proofs are equated modulo a natural notion of proof equivalence that computationally corresponds to the "true concurrency" of the computations.

Consider for example a system module `NIM` specifying a version of the game of Nim. There are two players and two bags of pebbles: a "draw" bag to remove pebbles from, and a "limit" bag to limit the number of pebbles that can be removed. The two players take turns making moves in the game. At each move a player draws a nonempty set of pebbles not exceeding those in the limit bag. The limit bag is then readjusted to contain the least number

of pebbles in either the double of what the player just drew, or what was left in the draw bag. The game then continues with the two bags in this new state. The player who empties the draw bag wins. An intermediate move is axiomatized by the rule `[mv]`; the last, winning move is axiomatized by the rule `win`.

```
mod NIM is
  protecting BOOL .
  sorts Pebble Bag State .
  subsorts Pebble < Bag .
  op o : -> Pebble .
  op nil : -> Bag .
  op __ : Bag Bag -> Bag [assoc comm] .
  op _=<_ : Bag Bag -> Bool .
  op least : Bag Bag -> Bag .
  op state : Bag Bag -> State .
  vars X Y Z : Bag .
  eq o nil = o .
  eq nil =< X = true .
  eq o X =< nil = false .
  eq o =< o  = true .
  eq o =< o X = true .
  ceq o X  =< o = false if X =/= nil .
  eq o X =< o Y = X =< Y .
  eq least(X,Y) = if X =< Y then X else Y fi .
  crl [mv] : state(X Y,Z) => state(Y,least(X X,Y))
                              if X =< Z and X =/= nil .
  crl [win] : state(X,Y) => state(nil,nil)
                             if X =< Y and X =/= nil .
endm
```

The initial model described by this module is the transition system containing exactly all the possible game moves allowed by the game. But there are many bad moves that would allow the other player to win. A good player should avoid such bad moves by having a *winning strategy*. With such a strategy, each move made by the player inexorably leads to success, no matter what moves the other player attempts.

What we obviously want, in this and in many other examples, is to have good ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate *strategies*. Many systems, for example theorem provers and declarative languages implementations, support certain strategies of this nature. However, such strategies are often *external* to the languages they control: they may constitute a separate programming language external to the logic, or may be part of the language's "extralogical features." In Maude, thanks to the reflective capabilities of rewriting logic, strategies can be made *internal* to rewriting logic. That is, they can be defined by rewrite rules, and can be reasoned about as with rules

in any other theory. The value of specifying strategies with rewrite rules is also emphasized in the most recent work on ELAN [2].

In fact, there is great freedom for defining 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 strategy language. Also, even if some users decide to adopt a particular strategy language because of its good features, such a language remains fully *extensible*, so that new features and new strategy concepts can be defined on top of them. Of course, such languages should be defined in a disciplined way that guarantees that they are correct, that is, that they only produce valid rewrites, as we explain below.

In Maude, a strategy language is a *function on theories*, that assigns to a module M another module $strat(M)$, whose terms are called *strategy expressions* specifying desired, possibly quite complex, set of rewrite deductions in the original theory M. Executing such a strategy expression is simply rewriting it using the rules in $strat(M)$. In some cases, such executions may never terminate. However, as the expression is being rewritten, more and more of the desired rewrites in the theory M that the strategy expression in question was supposed to describe become directly "visible" in the partially rewritten strategy expression. In this way, we can tame the wildness of M by shifting our ground to a much more controllable theory $strat(M)$. For example, $strat(M)$ may be Church Rosser, and therefore essentially a functional module, so that computations of strategy expressions become essentially deterministic. This is of course not a necessary requirement, but it is nevertheless an attractive possibility in the context of a sequential implementation.

We first briefly discuss reflection in rewriting logic and then explain how it can be used to define and give semantics to internal strategy languages. Rewriting logic is reflective [8,7]. That is, there is a rewrite theory $\mathcal{U}$ with a finite number of operations and rules that can simulate any other finitely presentable rewrite theory $\mathcal{R}$ in the following sense: given any two terms t, t' in $\mathcal{R}$ there are corresponding terms $\langle \overline{\mathcal{R}}, \overline{t} \rangle$ and $\langle \overline{\mathcal{R}}, \overline{t'} \rangle$ in $\mathcal{U}$ such that we have

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Let us denote by $FPTh$ the class of finitely presented rewrite theories. An *internal strategy language* is a theory-transforming function $strat : FPTh \longrightarrow FPTh$ that satisfies specific semantic requirements [8,7]. A sound methodology for defining such languages is to first define a *strategy language kernel* as a function, say, $meta : FPTh \longrightarrow FPTh$ that sends $\mathcal{R}$ to a definitional extension of $\mathcal{U}$—or a suitable subtheory of $\mathcal{U}$—by rewrite rules defining how rewriting in $\mathcal{R}$ is accomplished at the metalevel. A typical semantic definition that one wants to have in $meta(\mathcal{R})$ is that of $metaapply(\overline{l}, \overline{t})$, that simulates at the metalevel one step of rewriting at the top of a term t using the rule labeled l in $\mathcal{R}$. Proving the correctness of such a small strategy language kernel is then quite easy, by using the correctness of $\mathcal{U}$ itself as a universal theory. The next step is to define a strategy language of choice, say *strat*, as a function sending each theory $\mathcal{R}$ to a theory that extends $meta(\mathcal{R})$ by additional strategy expressions and corresponding semantic rules, all of which are recursive definitional extensions of those in the kernel in an appropriate sense, so that

their correctness can then be reduced to that of the kernel.

The descriptions of *meta* and *strat* that we have just given are phrased in metalevel terms, that is, they are described as metalevel functions. But in fact they are definable as functions within rewriting logic. Note that in $\mathcal{U}$ the theory $\mathcal{R}$ is represented as a *term* $\overline{\mathcal{R}}$. In fact, assuming a sorted version of the logic, all such terms $\overline{\mathcal{R}}$ are the elements of a sort `Module` in $\mathcal{U}$. This means that any effective function $F : FPTh \longrightarrow FPTh$ mapping a finitely presentable rewrite theory to another at the metalevel of the logic can now be represented at the object level as a computable function $\overline{F}$: `Module` $\longrightarrow$ `Module`. Therefore, by the metatheorem of Bergstra and Tucker [1], we can always specify such a function by a finite set of Church-Rosser and terminating rewrite equations in a suitable conservative extension of $\mathcal{U}$.

More details on the semantic definition of an internal strategy language for a logic in general, and for rewriting logic in particular, can be found in [7]. Since the rewrite engine can be naturally regarded as an implementation of key functionality in the universal theory $\mathcal{U}$, the Maude implementation supports a strategy kernel `META<X : Module>` in a built-in fashion for greater efficiency. The definition of a concrete strategy language `STRAT` as a functional module extending `META` is given in Appendix 9.

A strategy expression in `STRAT` initially has the form

```
rew T => ? with S
```

where `T` stands for the representation $\overline{t}$ in $\mathcal{U}$ of a term t in the object theory $\mathcal{R}$ in question—for example, the two pebble bag (`o o`) in `NIM` has the representation `'__['o,'o]` in `STRAT<NIM>`—and `S` is the rewriting strategy that we wish to compute. The symbol `?` indicates that we are beginning the computation of such a strategy; as the computation proceeds, `?` gets rewritten into a *tree of solutions*, and `S` is rewritten into the remaining strategy to be computed. In case of termination, this is the `idle` strategy and we are done.

This language can then be used to find a winning strategy for the `NIM` example. Such a strategy can easily be defined by extending the basic module `STRAT<NIM>` with a couple of mutually recursive strategies `movetowin` and `findawinner`

```
fmod NIM-WIN is
extending STRAT <NIM> .
ops mv win : -> Label .
ops movetowin findawinner : -> StrategyName .
vars T T' : Term . var SlT : SolTree . var SlTL : SolTreeList .

eq rew T => SlT{<- T'} with movetowin =
   rew T => SlT{<- T'} with
       (apply(win);; idle
        orelse (dk-apply(mv); findawinner)) .

eq rew T => SlT{<- mk(SlTL)} with findawinner =
   rew T => SlT{<- mk(SlTL)}
```

```
    with downleft ; (movetowin ;; (prunesol ; findawinner)
                     orelse (pruneresT ; up)) .

endfm
```

Intuitively, given a state $\langle X, Y \rangle$ in the game, `movetowin` will find a *winning move* $\langle X', Y' \rangle$ for a player A if there is one, in the sense that either $\langle X', Y' \rangle = \langle nil, nil \rangle$ or $\langle X', Y' \rangle$ is a move that eventually will lead the player A to success, no matter what moves the player B attempts, assuming that in the following moves, the player A always plays with the strategy `movetowin`.

In particular, `movetowin` defines the following strategy for a player A given a state $\langle X, Y \rangle$ in the game: try to win the game with just one move (`apply(win)`); if not, create a tree whose leaves $\langle X'_i, Y'_i \rangle$, $1 \leq i \leq n$, are all the allowed moves from the state $\langle X, Y \rangle$ (`dk-apply(mv)`). Then, try to find a leaf $\langle X'_i, Y'_i \rangle$ representing a state from which the player B can not make a winning move (`findawinner`); if not, the result of the strategy `movetowin` for the player A will be `failure`.

As expected, `findawinner` defines the following strategy for a player A over a tree T (possibly empty) of allowed moves: try to select the first leaf $\langle X'_1, Y'_1 \rangle$ of T (`downleft`); note that if T is empty, the result of `downleft` will be `failure`. Then, if the player B can make a winning move from $\langle X'_1, Y'_1 \rangle$ (`movetowin`), prune that leaf (`prunesol`) and try to find among the rest of the leaves a winning move (`findawinner`); if the player B can not make a winning move from $\langle X'_1, Y'_1 \rangle$, prune the rest of the leaves (`pruneresT`) and select $\langle X'_1, Y'_1 \rangle$ (`up`) as a winning move.

We can then run the following examples to find a winning move when there is one, or to fail to do so otherwise.

```
Maude>red rew 'state[('__['o,'o,'o,'o]),('__['o,'o,'o])] => ?
              with movetowin .
Result in sort StrategyExp:
      rew 'state[('__['o,'o,'o,'o]),('__['o,'o,'o])] =>
           ^{<- 'state['__['o,'o,'o],'__['o,'o]]} with idle .
Maude>red rew 'state[('__['o,'o,'o,'o,'o]),('__['o,'o,'o,'o])]
               => ? with movetowin .
Result in sort StrategyExp: failure .
```

5 Metaprogramming in Maude

Perhaps one of the most important new contributions of Maude is the *metaprogramming methodology* that it supports in a simple and powerful way. This methodology is well integrated with the language's semantic foundations, particularly with its logical foundations for reflection.

By "metaprogramming" we of course mean the capacity of defining programs that operate on other programs as their data; in our case, equational and rewrite theories that operate on other such theories as their data. By observing that we can not only reify theories, but also views among them, this

includes the more traditional "parameterized programming" capabilities in the Clear-OBJ tradition [4,11] as a particular instance. The difference is that in that tradition theories are metalevel entities not accessible at the object level of the logic, since this is only possible in an explicitly reflective logical context.

What reflection accomplishes is to *open up to the user* the metalelevel of the language, so that instead of having a fixed repertoire of parameterized programming operations we can now define a much wider range of theory-transforming and theory-combining operations that could not be defined using more traditional means. We have illustrated this power with the $meta(X : \texttt{Module})$ and $strat(X : \texttt{Module})$ constructions, that are "parameterized modules" in this much more general sense. Another good example, given in [15], is the reification of the logic map $\Psi : LLogic \longrightarrow RWLogic$ from linear logic to rewriting logic as an equationally defined function $\overline{\Psi} : \texttt{LLTheory} \longrightarrow \texttt{Module}$ inside rewriting logic. This example illustrates a general method by which, when using rewriting logic as a logical framework, we can always reify an effectively given map of logics $\Phi : \mathcal{L} \longrightarrow RWLogic$, sending finitely presentable theories in $\mathcal{L}$ to finitely presentable rewrite theories, as an equationally defined function $\overline{\Psi} : \texttt{Theory}_{\mathcal{L}} \longrightarrow \texttt{Module}$ inside rewriting logic.

Many more examples could be given. Indeed, we plan to systematically exploit Maude's metaprogramming capabilities to make the language and its environment very easily extensible and modifiable, and to support many logical framework and semantic framework applications such as: representation and interoperation of logics inside rewriting logic, executable definition of other logical languages in Maude, and definition of theorem-proving environments and tools for Maude and for other languages inside rewriting logic.

In summary, what reflection makes possible in Maude is the definition of an open, extensible, and user-definable *module algebra* supporting a new style of metaprogramming with very promising advantages for software methodology.

6 The Semantics of Maude

We summarize the semantic foundations of Maude's functional, object-oriented, and system modules.

6.1 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. The underlying equational logic chosen for Maude is *membership equational logic* [16,3], a conservative extension of both order-sorted equational logic and partial equational logic with existence equations [16]. It supports partiality, subsorts, operator overloading, and error specification.

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 nonerror, or defined, elements, whereas the elements without a sort are error or undefined elements.

Atomic formulas are either Σ-equations, or *membership assertions* 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 formulae, quantified by finite sets of K-kinded variables. That is, they are either conditional equations

$$(\forall X)\ \ t = t \ \ \textit{if} \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

or *membership axioms* of the form

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

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 [16].

In Maude, *functional modules* are equational theories in membership equational logic satisfying additional requirements. 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. *Functional theories* are also membership equational logic theories, but they have instead a loose interpretation, in that all models of the theory are acceptable, although a functional theory may impose the additional requirement that some of its subtheories should be interpreted initially. This is entirely similar to the treatment of "objects" and theories in OBJ [11]. Indeed, since membership equational logic conservatively extends order-sorted equational logic, Maude's functional modules extend OBJ modules.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. There is no need to declare kinds explicitly. The convenience of order-sorted notation is retained as syntactic sugar. Thus, an operator declaration

```
op push : Nat Stack -> NeStack .
```

is understood as the membership axiom

$$(\forall x, y)\ \ push(x, y) : NeStack \ \ \textit{if} \ \ x : Nat \wedge y : Stack.$$

Similarly, a subsort declaration `NeStack < Stack` corresponds to the membership axiom

$$(\forall x)\ \ x : Stack \ \ \textit{if} \ \ x : NeStack.$$

Computation in a functional module is accomplished by using the equations as rewrite rules until a canonical form is found. Therefore, the equations must satisfy the additional requirements of being Church-Rosser, terminating, and sort-decreasing [3]. 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. For a module satisfying such conditions any reduction strategy will reach a normal form; nevertheless, the user can assign to each operator a functional evaluation strategy in the OBJ style [11] to control the reduction for efficiency purposes. If no such strategies are declared, a bottom-up strategy is chosen. Since Maude supports rewriting modulo equational theories such as associativity or associativity/commutativity, all that we say has to be understood for equational rewriting *modulo* such axioms.

In membership equational logic the Church-Rosser property of terminating and sort-decreasing equations is indeed equivalent to the confluence of their critical pairs [3]. Furthermore, both equality and membership of a term in a sort are then *decidable* properties [3]. That is, the equality and membership predicates are *computable functions*. We can then use the metatheorem of Bergstra and Tucker [1] 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 memberships $not(t : s)$ 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 Curch-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.

Let us denote membership equational logic by $Eqtl^{:}$ and its associated rewriting logic by $RWLogic^{:}$. Regarding an equational theory as a rewrite theory whose set of rules is empty defines a conservative map of logics [15]

$$Eqtl^{:} \longrightarrow RWLogic^{:}$$

This is the way in which Maude's functional modules are regarded as a special case of its more general system modules.

6.2 Semantics of object-oriented and system modules

As already pointed out, the logic of Maude is the membership logic variant of rewriting logic $RWLogic^{:}$. A *system module* is then a rewrite theory. In the unparameterized case its semantics is the initial model defined by the theory [19], which is the algebra of all rewriting computations for ground terms in the theory. From a systems perspective this model describes all the concurrent behaviors that the system so axiomatized can exhibit. From that perspective a term t denotes a state of the system, and a rewrite $t \longrightarrow t'$ denotes a possibly concurrent computation.

A system module can contain one or more parameter theories. The inclusion from the parameter(s) into the module then gives rise to a free extension functor [18], which provides the semantics for the module. This of course means that we can compose systems by putting together the rewrite theories

in which they are specified.

A rewrite theory has both rules and equations, so that rewriting is performed *modulo* such equations. However, this does not mean the Maude implementation must have a matching algorithm for each equational theory that a user might specify, which is impossible, since matching modulo an arbitrary theory is undecidable. What we instead require for theories in system modules is that:

- The equations are divided into a set A of axioms, for which matching algorithms exist in the Maude implementation [4], and a set E of equations that are Church-Rosser, terminating and sort decreasing *modulo* A; that is, the equational part must be equivalent to a functional module.
- The rules R in the module are *coherent* [26] (or at least what might be called "weakly coherent" [20], Section 5.2.1) with the equations E modulo A. This means that appropriate critical pairs exist between rules and equations allowing us to intermix rewriting with rules and rewriting with equations in any way 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. In particular, a simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R.

Since the state of the system specified by a system module is axiomatized as an abstract data type by the equations E modulo A, and the rules in R are local rules for changing such a state, in practice the lefthand sides of rules in R only involve constructor patterns, so that coherence is a natural byproduct of good specification practice. Besides, using the completion methods in [26] one can check coherence, and one can try to make a set of rules coherent when they are not so.

The semantics of object-oriented modules is entirely reducible to that of system modules, in the sense that there is a systematic desugaring process translating each object-oriented module into its corresponding system module [20]. However, the particular ontology supported by object-oriented modules is something very much worth keeping, and it does not exist for general system modules. 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. 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 [20,21] provide good background on such foundations. Talcott's paper [25] gives rewriting logic

[4] 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 [9]. At present, matching modulo associativity and commutativity, and a preliminary version of matching modulo associativity are supported.

foundations for actors from a somewhat different viewpoint.

The basic ideas about the reflective semantics of Maude have already been discussed in Section 4. Much more detail can be found in [7].

7 The Maude Implementation

This section describes the implementation of the Maude interpreter, which consists of two main components: the front end and the engine.

7.1 Front end and module evaluation

The front end of the Maude interpreter is built on top of the OBJ3 front end, and is written in Common Lisp. The Maude front end shares with OBJ3 the convenient mixfix syntax for user-defined symbols and expressive parameterized programming mechanisms. The Maude front end augments this with additional syntax for Maude language constructs, tracing and debugging commands, complete disambiguation of ad-hoc overloaded operators, a complete module-flattening operation, a specialized pretty- and unpretty-printer, a program transformation from object-oriented modules to system modules, and support for meta-level specifications. The result is that users can enter Maude specifications using powerful parameterized programming constructs and mixfix syntax which are completely eliminated before a Maude specification is passed to the engine. Output from the engine is passed back through a pretty-printer which reparses the output in prefix form, and then prints the result in the user-declared mixfix style. Timing and rewriting statistics from the engine are also reported from the engine to the user through the front end.

7.2 Maude's rewrite engine

The design objectives of the Maude rewrite engine are consistent with the executable specification and formal method uses that we wish to support. The system should "look and feel" like an interpreter, should be capable of supporting user interrupts and source level tracing, and above all should be extensible with new equational theories and new built-in operators both of which may require new term/data representations to be integrated seamlessly with existing term/data representations. Reflective capabilities are also central to our design, since the system should support arbitrary levels of meta-rewriting.

Although we have sought the most efficient implementation meeting the above objectives, supporting them all but rules out a number of performance enhancing techniques such as: compilation to native machine code (or 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—i.e., where code operating on symbols in one equational theory recognizes symbols in alien theories and makes use of their properties.

The design chosen is essentially a highly modular semi-compiler where the most time consuming run-time tasks are compiled at parse-time into a sys-

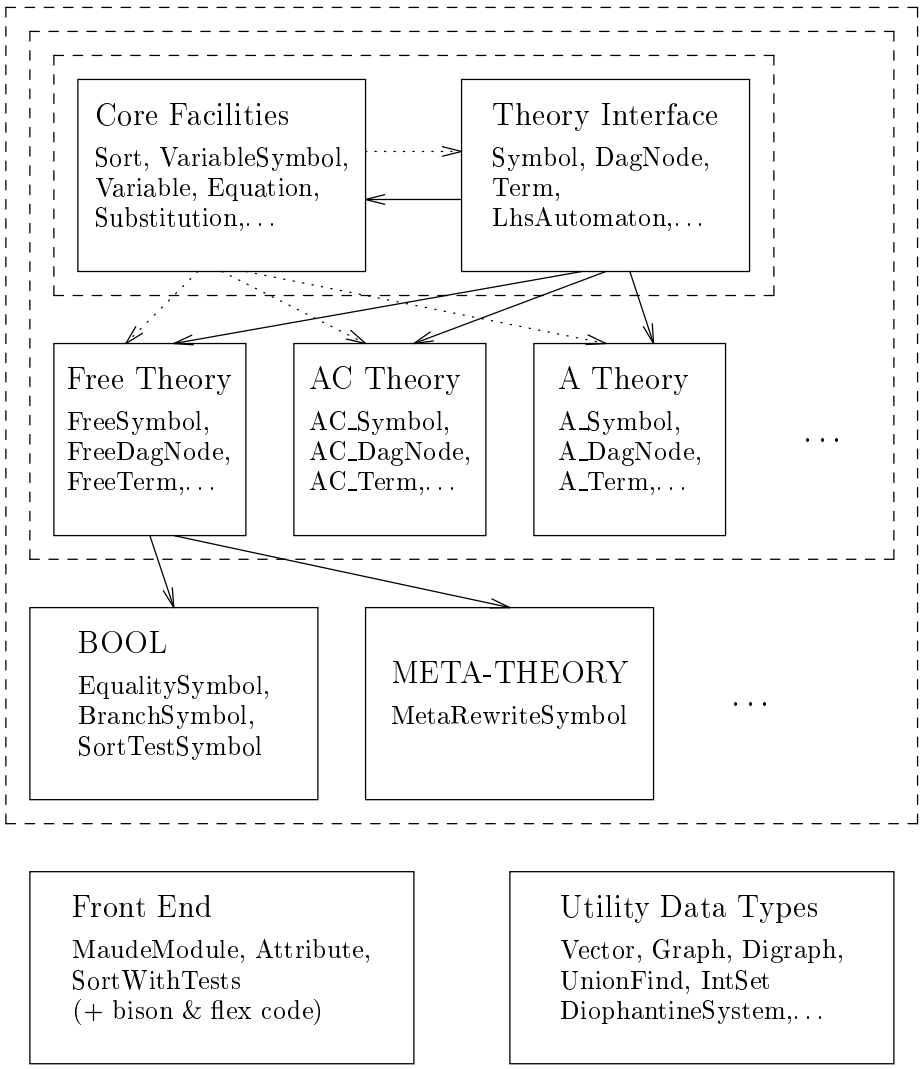

Fig. 1. Overall structure of the Maude Interpreter's Rewrite Engine

tem of lookup tables and automata which are interpreted at run-time. After some early experiments it was found very useful to have two distinct representations for terms. For most uses terms are represented as trees, in which nodes are decorated with all kinds of information to simplify parse time analysis. For the subject term being rewritten, however, a directed-acyclic-graph (DAG) representation is used with very compact nodes. Heavy use is made of object-oriented structuring techniques and great care has been taken to ensure extensibility and to make the bulk of the engine application-independent.

The overall structure of the rewrite engine is shown in Figure 1, where each module is shown as a box and some of the names of the modules classes are shown in each box.

Solid arrows indicate that some of the classes in the target module are derived from classes in the source module; dotted arrows indicate that classes in the target module use facilities provided by the source module. The modules themselves are organized in a layered structure where inner layers have no knowledge of, or dependency on, outer layers.

The innermost layer consists of the modules *Core Facilities* and *Theory Interface*. The Theory Interface consists of abstract classes for basic objects whose concrete realization will differ for different equational theories, such as: symbols, dag nodes, terms, lefthand side automata (for matching), righthand side automata (for constructing and normalizing righthand side and condition instances), matching subproblems and matching extension information. Some of the classes in the Theory Interface contain some concrete data and function members to provide useful common functionality to derived classes. The Core Facilities module consists of concrete classes for basic objects that are independent of the different equational theories, such as: sorts, connected components (kinds), variable symbols, variables (as terms), equations, sort constraints, rules, sequences of matching subproblems and substitutions. Neither the Core Facilities nor the Theory Interface treat any sort, symbol or equational theory as special in any way whatsoever; all are manipulated through virtual functions in the abstract classes belonging to the Theory Interface. In particular, this means that the code that handles conditional equations knows nothing about the Maude built in sort *Bool* and its built in constants *true* and *false*. Instead conditional equations always have the form

$$l = r \text{ if } c_1 = c_2.$$

and if a more complex boolean condition b is desired, it is encoded as the equality $b = \mathit{true}$.

The next layer consists of modules for individual equational theories. Each module in this layer consists of concrete descendents of abstract classes from the Theory Interface, which provide a theory-specific implementation of virtual functions such as *match*(), *compileLhs*() and *rewrite*(). In this way each equational theory has its own representation objects such as symbols, terms, dag nodes and matching automata. At this level there are no special sorts or symbols and each module is only aware of the representation of its own classes; everything else is *alien* and is manipulated through the Theory Interface.

The next layer consists of modules containing classes which provide symbols with non-standard run-time properties. Even here there are no special sorts or symbols; only classes for symbols that have rather generalized non-standard run-time behavior. The *BranchSymbol* class for example can be used to generate all manners of conditional constructs including the 'if-then-else-fi' needed for Maude. These classes only affect the behaviour of a symbol when an attempt is made to rewrite at a dag node containing it. All other properties (such as matching and normalization) and data representations are inherited from the parent equational theory.

The outermost module *Front End* contains a rudimentary parser, the class *MaudeModule* and a couple of minor classes. Only here do Maude specific operators such as 'if-then-else-fi' and 'meta-apply' really exist. The Front End is dependent on all the other modules but no other module depends on it. It can be changed or replaced without modifying the rest of the engine.

One final module is the *Utility Data Types*. This contains classes and class templates implementing 'components of general utility' such as vectors, graphs

and Tarjan's union-find data structure. These are used freely throughout the engine.

Performance enhancing techniques implemented in the current prototype include:

(i) Fixed size dag nodes for in-place replacement.

(ii) Full indexing for the topmost free function symbol layer of patterns; when the patterns for some free symbol only contain free symbols this is equivalent to matching a subject against all the patterns simultaneously.

(iii) 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 conditions (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.

(iv) Use of binary search during AC matching for fast elimination of ground terms and previously bound variables.

(v) Use of a specially designed sorting algorithm which uses additional information to speed up the renormalization of AC terms.

(vi) Use of a Boyer-Moore style algorithm for matching under associative function symbols.

(vii) Compile time analysis of sort information to avoid needless searching during associative and AC matching.

(viii) Compile 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.

(ix) Compile time allocation of fixed size data structures needed at run time.

(x) Caching dynamically sized data structures created at run time for later reuse if they are big enough.

(xi) Bit vector encoding of sort information for fast sort comparisons.

(xii) Compilation of sort information into *regularity tables* for fast incremental computation of sorts at run time.

(xiii) Efficient handling of *matching with extension* through a theory independent mechanism that avoids the need for extension variables or equations.

In large examples involving the free theory, we have observed speedups in the order of 35–55 times faster than the OBJ3 implementation, reaching up to 200,000 rewrites per second on a 90 MHz Sun HyperSPARC. For examples of associative commutative rewriting we have observed typical speeds of 10,000 rewrites per second, and in some cases three or more orders of magnitude speedup over OBJ3.

The current version of the engine comprises 79 classes implemented by approximately 19500 lines of C++.

8 Future Plans

We have introduced the main ideas and the basic principles of Maude and have illustrated them with examples. In addition to continued work on theoretical foundations much more experimentation and implementation work lies ahead of us. The following areas will receive special attention:

- Further development of, and experimentation with, Maude's reflective and metaprogramming capabilities.
- Experimentation with different strategy languages, development of useful strategy libraries, and study of parallel strategies.
- Extension of the rewrite engine with matching algorithms for new equational theories.
- Implementation of unification algorithms to support narrowing computations in addition to rewriting. This will also allow adequate treatment of rules with extra variables in their righthand sides, that are not supported by the current implementation.
- Development of a theorem-proving environment supporting automated reasoning about specifications in Maude and in other languages.
- Implementation of foreign interface modules [24,20], to support frequently occurring computations in a more efficient, built-in way.
- Input-Output. This should be naturally specified using Maude's concurrent object-oriented concepts.
- Compilation of Maude, as well as parallel and distributed implementations of the language.
- Applications and case studies. Application areas that seem particularly promising include: logical framework applications, module algebra and metaprogramming methodology, object-oriented applications, symbolic simulation, real-time system specification, parallel programming, and uses of Maude as a programming language definition and prototyping tool.

Acknowledgements
We cordially thank Timothy Winkler and Narciso Martí-Oliet for their valuable contributions to the development of the Maude ideas. We also thank Carolyn Talcott for many discussions on Maude and for her valuable suggestions on strategy aspects. We are grateful for very helpful discussions and exchanges with Kokichi Futatsugi, Claude and Hélène Kirchner, Martin Wirsing, Ulrike Lechner, Christian Lengauer, and many other colleagues. Our previous work with Joseph Goguen and the other members of the OBJ team has also influenced the development of our ideas.

References

[1] Jan Bergstra and John 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*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

[2] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. This volume.

[3] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. Manuscript, SRI International, August 1996.

[4] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer LNCS 86, 1980.

[5] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[6] A. Ciampolini, E. Lamma, P. Mello, and C. Stefanelli. Distributed logic objects: a fragment of rewriting logic and its implementation. This volume.

[7] Manuel G. Clavel and José Meseguer. Reflection and strategies in rewriting logic. This volume.

[8] Manuel G. Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

[9] Steven Eker. Fast matching in combination of regular equational theories. This volume.

[10] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.

[11] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992.

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

[13] Simon S. Lam and A. Udaya Shankar. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, SE-16(7):755–775, July 1990.

[14] Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch, K.M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.

[15] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[16] José Meseguer. Membership algebra. Lecture at the Dagstuhl Seminar on "Specification and Semantics," July 9, 1996. Extended version in preparation.

[17] José Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.

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

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

[20] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[21] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In Oscar M. Nierstrasz, editor, *Proc. ECOOP'93*, pages 220–246. Springer LNCS 707, 1993.

[22] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proceedings of the CONCUR'96 Conference, Pisa, August 1996*. Springer LNCS, 1996.

[23] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, 1993.

[24] José Meseguer and Timothy Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.

[25] C. L. Talcott. An actor rewrite theory. This volume.

[26] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.

9 Appendix

```
fmod META <M : Mod> is
sorts OpId VarId Term TermList Label Nat .
```

```
subsort VarId < Term .
subsort OpId < Term .
subsort Term < TermList .

op _[_] : OpId TermList -> Term .
op _,_ :  TermList  TermList -> TermList [assoc] .
op error* : -> Term .
*** meta-apply is a built in function, that takes the meta-representation
*** of a term, a rule label and a natural number in peano representation.
***
*** meta-apply(t, l, n) is evaluated as follows:
*** (1) "t" is converted to the term it represents.
*** (2) this term is fully reduced using the equations
*** (3) the resulting term is matched against all rules with label "l"
***     with matches that fail to satisfy the condition of their rule
*** discarded.
*** (4) the first "n" successful matches are discarded
*** (5) if there is an (n+1)th match, its rule is applied using that
***     match; otherwise "error*" is returned
*** (6) the new term is fully reduced using the equations
*** (7) the resulting term is converted to a meta-term which is returned
op meta-apply : Term Label Nat -> Term .
op z : -> Nat .
op s : Nat -> Nat .
endfm

*** Here we just introduce the specification of STRAT <M : Mod> needed to
*** compute reductions in NIM-WIN
fmod STRAT <M : Mod> is
extending META <M> .
sorts SolTree SolTreeList SolTreeExp StrategyName Strategy StrategyExp .
subsort Term < SolTree .
subsort SolTree < SolTreeList .
subsort SolTree < SolTreeExp .
subsort StrategyName < Strategy .

op ? : -> SolTreeExp .
op ^ : -> SolTree .
op _,_ :  SolTree SolTreeList -> SolTreeList .
op mk :  SolTreeList -> SolTree .
op _{<-_} : SolTree SolTree -> SolTree .
op sols : Term Label Nat -> SolTreeList .
op failure : -> StrategyExp .
op rew_=>_with_ : Term SolTreeExp Strategy -> StrategyExp .
op _andthen_ : StrategyExp Strategy -> StrategyExp .
op idle : -> Strategy .
op _;_ : Strategy Strategy -> Strategy .
op _;;_orelse_ : Strategy Strategy Strategy -> Strategy .
op apply : Label -> Strategy .
```

```
op dk-apply : Label ->  Strategy .
op downleft : -> Strategy .
op up : -> Strategy .
op prunesol : -> Strategy .
op prunerest : -> Strategy .

var N : Nat . vars T T' T'' : Term .  var L : Label .
var SlT SlT' : SolTree . var SlTL : SolTreeList .
var S S' S'' : Strategy .

eq rew T => ? with S = rew T => ^{<- T} with S .
eq rew T => SlT with (S ; S') = (rew T => SlT with S) andthen S' .

eq rew T => SlT with idle andthen S = rew T => SlT with S .
eq failure andthen S = failure .

eq rew T => SlT with (S ;; S'' orelse S') =
   if rew T => SlT with S == failure then rew T => SlT with S'
     else rew T => SlT with S andthen S'' fi .

eq rew T => SlT{<- T'} with apply(L) =
   if meta-apply(T',L,z)== error* then failure
     else rew T => SlT{<- meta-apply(T',L,z)} with idle fi .

eq rew T => SlT{<- T'} with dk-apply(L) =
   rew T => SlT{<- mk(sols(T',L,z))} with idle .

eq rew T => SlT{<- mk(sols(T',L,N))} with downleft =
   if meta-apply(T',L,N) == error* then failure
   else rew T => SlT{<- mk(^,sols(T',L,s(N)))}
               {<- meta-apply(T',L,N)} with idle fi .

eq rew T => SlT{<- mk(^,SlTL)}{<- T'} with prunesol =
   rew T => SlT{<- mk(SlTL)} with idle .

eq rew T => SlT{<- mk(^,SlTL)}{<- T'} with pruneresT =
   rew T => SlT{<- ^}{<- T'} with idle .

eq rew T => SlT{<- ^}{<- T'} with up = rew T => SlT{<- T'} with idle .
endfm
```