Electronic Notes in Theoretical Computer Science 36 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume36.html 22 pages

Towards Maude 2.0 [★]

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] *ETSII, 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 2.0 is the new version of the Maude rewriting logic language currently under development. Maude 2.0's three main goals are: (i) greater generality and expressiveness; (ii) efficient support for a wider range of programming applications; and (iii) usability as a key component for developing internet programming and mobile computing systems. To meet these goals, a number of new features have been added. The membership equational logic of functional modules and the rewriting logic of system modules are now supported in their greatest possible generality, and the operational semantics of object-oriented modules guarantees object and message fairness. Module operations in Full Maude are also more general thanks to parameterized theories and views. Efficient support for a wider range of programming applications is provided both by the Maude compiler—which can reach up to 15 million rewrites per second on a 667MHz Xeon—and by a library of new built-in modules. Besides new built-in functional modules, a key new feature is built-in object-oriented modules that provide flexible interaction with external objects such as file systems, window systems, and internet sockets. In particular, built-in internet sockets will provide excellent support for a new declarative style of internet programming in Maude, and will be used as a key building block to implement the Mobile Maude language.

[★] Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by Office of Naval Research Contract N00014-99-C-0198, and by National Science Foundation Grants CCR-9505960 and CCR-9900334.

1 Introduction

Maude 2.0 is the new version of Maude currently under development. This paper presents our detailed design of the key features of Maude 2.0, some of which have already been implemented. Maude 2.0's three main thrusts are: (i) greater generality and expressiveness; (ii) efficient support for a wider range of programming applications; and (iii) usability of Maude 2.0 as a key component for developing internet programming and mobile computing systems. The paper explains the design decisions we have made in support of these three main objectives.

Section 2 focuses on the new module syntax and semantics supporting greater generality and expressiveness in specifications. Essentially, the underlying logics—membership equational logic for functional modules, and the version of rewriting logic extending it for system modules—will be supported in their greatest possible generality. Similarly, parameterized modules, theories and views will support a greater degree of modularity and reusability through parameterized theories and parameterized views. The operational semantics of all modules will likewise be extended. In particular, extra variables will be allowed in conditions which, for conditional rules, may include rewrite conditions; also, object-oriented modules will be executed in an object- and message-fair way by the default interpreter for rules.

Efficient support of many programming applications requires an adequate library of built-in modules and advanced compilation technology. These two topics are addressed, respectively, in Sections 3 and 6. Besides a number of new built-in functional modules, the most important of which is *strings*, a key novelty is the addition of *built-in object-oriented modules*. Such modules will be seamlessly integrated with ordinary object-oriented modules. In this way, ordinary Maude objects will be able to communicate and interact by message passing with built-in objects such as file systems, window systems, and internet sockets. This will greatly extend the range of applications that can be supported, and will provide greater flexibility for programming interactive applications. Built-in internet sockets will be a key feature allowing the use of Maude as an internet programming language, and will support the implementation of the Mobile Maude language [9] on top of Maude. Section 6 explains the Maude compiler, which is already quite advanced and efficient, and that will be integrated with the Maude 2.0 interpreter.

Any changes and advances to the language must be reflected in the module `META-LEVEL`. Section 4 explains how this will be done, while at the same time achieving a greater simplicity and economy in the metarepresentation of terms and modules, and providing a richer set of metalevel operations. Two other miscellaneous features are gathered in Section 5, namely, the new module system, allowing a greater efficiency in specifications involving large module hierarchies, and the `latex` attribute, that can be declared for any operator in a module to output module declarations and evaluated expressions in the desired LaTeX mathematical notation. The paper ends with some conclusions in Section 7. The appendices give some more details about some built-in modules, and present a sample internet programming application based on built-in internet sockets.

The paper assumes some familiarity with the basic concepts of Maude, such as its logical basis in membership equational logic [2] and rewriting logic [17], its functional, system, and object-oriented modules, and module operations, and its reflective features, for which we refer the reader to [7,6].

2 New Module Syntax and Semantics

This section discusses the extensions to the current syntax and semantics of functional, system and object-oriented modules, and of parameterized modules and theories that will be supported in Maude 2.0.

2.1 Functional Modules

The new syntax for functional modules extends the previous syntax to allow the greatest possible generality in the support of membership equational logic specifications, and offers a more uniform syntax for variables. First of all, to support specification of partial operations that do not restrict to a total function on a product of sorts, we allow explicit declaration of such functions at the kind level. In membership equational logic, terms that have a kind but not a sort are understood as *undefined* or *error* terms. A *kind* k has a family S_k of associated *sorts*, which are semantically understood as subsets. In general, a total function at the kind level restricts only to a *partial* function at the level of sorts.

In functional modules, kinds are not explicitly named. Instead, we identify a kind k with the set S_k of its sorts, understood 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. This provides a very robust way of referring to equivalence classes when a module is imported into another module, which may increase the number of sorts in a class and may even merge several classes. For example, if a new supersort s' of a previous sort s is introduced in a new module, then, $[s]$ and $[s']$ are two equivalent ways of referring to the *same* kind.

Consider for example the concatenation function for paths in a graph. It is really a partial function. It could be made total on a user-defined supersort `Path?` of `Path`, but it is simpler and more elegant to define it at the kind level by a declaration

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

A second syntax extension regards the treatment of *variables*. A variable is now an identifier composed of a name, followed by a colon, followed by a sort or kind name. For example, `P:Path` is a variable of sort `Path`. In this way, variables do not have to be declared in variable declarations: they can appear directly in terms. Variable declarations are still allowed for convenience, but a declaration

```
vars P Q R S : Path .
```

is now understood as an *alias definition*, allowing, for example, using the name `P` as an abbreviation for the variable `P:Path`.

A third syntax extension regards the treatment of *equational conditions* in conditional equations and conditional memberships. Such conditions are *conjunctions of equations and memberships*, and they 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'`. Both have the same equational interpretation in the mathematical semantics, but a different operational semantics, as explained below. For example, assuming the above variable declaration, the associativity of path concatenation can be expressed by the conditional equation

```
ceq (P ; Q ); R = P ;(Q ; R) if t(P) = s(Q) /\ t(Q) = s(R) .
```

where `s` and `t` are respectively the source and target functions. Note that there is no need to express the condition as a single *Boolean condition*, using the Boolean-valued equality predicate `==`. This is still possible, namely, by declaring

```
ceq (P ; Q ); R = P ;(Q ; R) if t(P) == s(Q) and t(Q) == s(R) .
```

but is not necessary. 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`.

We can illustrate the above-mentioned extensions by means of the following functional module `PATH`, for paths in a graph, that imports a module `A-GRAPH` with sorts `Node` and `Edge`, operations `s` and `t`, giving the source and target nodes of each edge, and specific edge and node constants that need not concern us here (see an example in Appendix A.1). It then builds paths on such a graph according to the following equations and memberships.

```
fmod PATH is protecting A-GRAPH .
  sort Path .
  subsorts Edge < Path .
  op _;_ : [Path] [Path] -> [Path] .
  ops s t : Path -> Node .
  var E : Edge .   vars P Q R S : Path .
  cmb E ; P : Path if t(E) = s(P) .
  ceq s(P) = s(E) if E ; S := P .
  ceq t(P) = t(S) if E ; S := P .
  ceq (P ; Q ); R = P ;(Q ; R) if t(P) = s(Q) /\ t(Q) = s(R) .
endfm
```

Note the matching equation `E ; S := P` in the conditions of the two conditional equations defining the source and target functions. As mentioned before, 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.* In general, 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. In the above example, the many-kinded unifier $\{\texttt{E} \longleftarrow \texttt{P ; Q}, \texttt{S} \longleftarrow \texttt{R}\}$ obtained by unification with the lefthand side of the associativity equation is ruled out by the fact that the membership `P ; Q : Edge` is unsatisfiable.

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

$$t = t' \;\; \mathit{if} \;\; C_1 \wedge \ldots \wedge C_n$$

in a functional module M have to satisfy the following *admissibility requirements*[1], ensuring that all the extra variables will become instantiated by matching:

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

(ii) If C_i is an equation $u_i = u_i'$ or a membership $u_i : s$, then
$$\mathit{vars}(C_i) \subseteq \mathit{vars}(t) \cup \bigcup_{j=1}^{i-1} \mathit{vars}(C_j).$$

(iii) If C_i is a matching equation $u_i := u_i'$, then u_i is an M-pattern and
$$\mathit{vars}(u_i') \subseteq \mathit{vars}(t) \cup \bigcup_{j=1}^{i-1} \mathit{vars}(C_j).$$

[1] These requirements include as a special case what are called *properly oriented and right stable 3-CTRSs* in [21], when each equation $s_i = t_i$ in their conditions is expressed as a matching equation $t_i := s_i$.

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

In spite of the added generality of allowing extra variables in the conditions, we still expect functional modules to be Church-Rosser and terminating membership equational logic specifications in the sense of [2, Section 10.1]. The above admissibility requirements and the Church-Rosser and termination assumptions are dropped for functional *theories* (see Section 2.4) which support the full generality of the logic.

For functional modules Maude does indeed support the operational semantics by rewriting defined in [2], generalized to allow extra variables as sketched in Section 10.1 of [2]. In particular, the previous *strictness* assumption that the substitution instance of a left-hand side of an equation has a sort (see [6, Section 6.1]) is dropped in Maude 2.0 to allow more general, nonstrict equations such as those used for error recovery. When strictness is desired, the equations must have explicit conditions ensuring it.

2.2 System Modules

At the equational level, system modules support also all the extensions and 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 now take the most general possible form in the variant of rewriting logic built on top of membership equational logic. That is, they can be of the form

$$t \longrightarrow t' \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

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. Finally, in spite of the full generality allowed for conditional rewrite rules, we still expect the rewrite rules to be *coherent* or *weakly coherent* [23] with respect to the equations in the module, so that the equational and the rewriting parts of the deduction can be efficiently modularized.

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 [3,22]. However, a quite general class of system modules, called *admissible modules*, will be executable by Maude 2.0's default interpreter. As already mentioned, the equational part of a system module must always satisfy the same requirements given in Section 2.1 for functional modules. A system module M is called *admissible* if, in addition, each of its rewrite rules

$$t \rightarrow t' \;\; if \;\; C_1 \wedge \ldots \wedge C_n$$

satisfies the admissibility requirements (i)-(iii) in Section 2.1 plus the additional requirement

(iv) If C_i is a rewrite $u_i \rightarrow u_i'$, 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 $v_i \rightarrow w_i$ 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. Therefore, the default interpreter will support search computations, and will allow controlling the search by means of adequate parameters. In general, the goals 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 the new syntax for system modules by means of an admissible module from [16] that defines the transition system semantics for Milner's CCS [20] in such a way that transitions correspond to rewrites. Full CCS is represented, including (possibly recursive) process definitions by means of contexts. The reader can find the modules defining the syntax in Appendix A.2.

```
mod CCS-SEMANTICS-TRANS is protecting CCS-CONTEXT .
   sort ActProcess .
   subsort Process < ActProcess .
   op {_}_ : Act ActProcess -> ActProcess .
      *** {A}P means that process P has performed action A
   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) = true  /\  (A =/= ~ L) = true .

   *** 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) = true /\  (A =/= ~ L) = true .

   *** Definition
   crl X => {A}P' if (X definedIn context) = true
                  /\  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 [16]).

2.3 Object-Oriented Modules

Many Maude applications make essential use of *object-oriented modules* (see for example the survey papers [8,19]), that is, modules specifying distributed object systems in which the top structure of the distributed state is an associative and commutative *multiset* of *objects* and *messages.* Default execution of object-oriented modules is the most common way of using such modules for both specification and programming. That is, although it is very useful to formally analyze object-oriented specifications using appropriate strategies, it is also very important to support well their default execution.

The problem is that the general notion of *fairness* supported by the previous default interpreter is insufficient in the object-oriented case. 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. Therefore, the Maude 2.0 default interpreter will support a special strategy for execution of object-oriented modules that ensures both object and message fairness.

The intuitive idea of such a fair execution strategy can be explained by means of a metaphor in which we think of objects and messages as entities that, like cars, need *gas* to run. The default strategy then refills such entities with gas from time to time, and any rewrite decreases the amount of gas, but no new refills are allowed until no more progress can be made. In this way, all objects and messages in a configuration are repeatedly given a fair chance to be rewritten.

This default strategy for objects has an important additional advantage, namely, that using built-in objects such as internet sockets, file systems, or window systems (see Section 3.2) it is then very easy to execute object-oriented configurations consisting of both ordinary and built-in objects in a truly concurrent and fair way.

2.4 Parameterized Modules and Theories

The Full Maude 1.0 module algebra supports quite powerful module composition operations in the Clear/OBJ style. Such operations use categorical constructions involving three key entities:

- *modules*, which are theories with an *initial* or—in the parameterized case—*free extension* semantics;
- *theories*, with a *loose* semantics, that can be used to specify the parameters of modules and to state formal assertions; and
- *views*, which are theory interpretations used to instantiate parameter theories, refine specifications, and assert formal properties.

It has long been understood that the full generality and power of a module algebra based on these primitives requires *parameterized* theories and views, not just parameterized modules. However, at present, neither OBJ3, nor CafeOBJ, nor Maude 1.0 support parameterized theories and views. This situation will be remedied in Maude 2.0 by means of the Full Maude 2.0 module algebra, in which modules, theories, and views can all be parameterized.

By using parameterized theories and views we can instantiate parameterized theories and modules in a more incremental way, gaining in flexibility and being able to specify precisely the intended applications. The use of parameterized views will allow us, for example, to define a view `Set[X :: TRIV]` from `TRIV` to `SET[X :: TRIV]` mapping the sort `Elt` to the sort `Set[X]`. With this kind of views we keep the parameter part of the target module still as a parameter. For example, given the view `Nat` from `TRIV` to `NAT`, we can have the module `LIST[Set[Nat]]` of lists of sets of natural numbers, or stacks of sets of booleans with `STACK[Set[Bool]]` given a module `STACK[X :: TRIV]` of stacks and a view `Bool` from `TRIV` to the built-in module `BOOL`.

For more details we refer the reader to the companion paper [10], in which the key design concepts and constructs of Full Maude 2.0 are explained and illustrated with examples.

3 Built-in Modules

Maude includes some built-in modules providing convenient high-performance functionality within the Maude system, as well as effective connections to other systems of interest. In particular, Maude 2.0 will provide built-in machine integers, natural and floating-point numbers, quoted identifiers, strings, and internet sockets. We are also in the process of designing potential built-in modules for rational numbers, file systems, and window systems.

3.1 Functional Built-in Modules

The functional built-in modules of machine-integers, natural and floating-point numbers, quoted identifiers, and strings provide a minimal set of efficient implementation constructs for Maude programmers.

Machine integers allow Maude programmers to deal with fixed-length binary representations of numbers between machine-specific minimum and maximum numbers. Operations on machine integers employ the constant-time machine operations on those binary representations. C-like performance for simple arithmetic operations on machine integers is achievable in Maude through the use of such built-in module. Built-in natural numbers bridge the gap between clean Peano-like axiomatizations of numbers with an explicit successor function, and rather more efficient binary representations of unbounded natural number arithmetic. This built-in module will allow programmers to manipulate numbers as if they were represented with explicit successor notation, and to reflect those numbers up to the meta-level (and meta-meta-level, etc). The module of floating-point numbers allows Maude users access to the IEEE-754 double precision floating-point arithmetic when this is supported by the underlying hardware platform.

Quoted identifiers allow Maude users to create new names and manipulate structures containing symbols in an efficient way. Built-in strings are the area of most recent work. Built-in strings, in particular standard strings of 8-bit characters, are essential to achieve the kinds of performance and expressiveness required for internet programming, file handling, and certain input-output procedures. In this current incarnation, Maude's built-in strings utilize only the 8-bit character variant. In the future, it may be desirable to extend this coverage to other character set conventions such as Unicode. Maude's built-in strings are based on the SGI rope package [1] which has been optimized for functional programming, where copying with modification is supported efficiently, while arbitrary in-place updates are not.

A *string identifier* is a Maude identifier which contains exactly one pair of matching `"`s which are its first and last characters. When a string identifier is parsed as a string it is interpreted using a subset of ANSI C backslash escape conventions [13, Section A2.5.2]. Strings of length one form a subsort `Char` of `String`. The current implementation of built-in strings includes the functions described briefly in Appendix B. The Maude string package is compatible with the `QID` built-in module [5], and interoperates with the Maude 2.0 scheme for meta-representing user constants.

3.2 Object-Oriented Built-in Modules

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. In previous versions of Maude, except for the textual interaction supported by the `LOOP-MODE` module [5], all interactions with objects could be specified and simulated within Maude, but could not be performed in the external world. In Maude 2.0, 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 can be defined in *built-in object-oriented modules* that define 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.

The part containing the built-in objects contains also a *system object*, or proto-object [18, Section 4.4], which takes care of answering requests for creation of new built-in objects and of sending the new object's name to the requesting object, and also of answering class-specific queries[2]. In general, the two-way traffic of messages between the configuration of ordinary objects and messages and the built-in objects consists of:

(i) requests for built-in object creation, and answers providing the new name to the requester,

(ii) class-specific queries to the system object, and corresponding answers, and

(iii) messages from ordinary objects to built-in objects, and reply messages in the other direction.

We illustrate these ideas below with the class of built-in internet sockets currently under implementation.

[2] The system class is imported by all built-in classes, but it can be further specialized by subclassing to provide answers to class-specific queries.

3.3 Built-in Internet Sockets

Perhaps the most interesting of the object-oriented built-in modules is the internet socket interface, intended to support internet programming. Internet computing poses different problems for the programmer and the language implementor than traditional sequential computing. In particular, since there is a thousand-fold difference between typical processing speed and typical internet connections, issues such as nonblocking access to resources and native notions of fairness are much more important than raw speed or other language choice considerations.

Maude is exceptionally well suited to enable internet programming. The native and semantically clean approach to concurrency in Maude provides a clean foundation. The inherently fair non-blocking nature of rewriting logic specifications, and of any compliant implementation, provides a welcome alternative to clumsy threaded implementations in sequential languages in the internet environment. Effectively, Maude's design inherently provides multiplexing of network communications.

Internet programming is done via a suite of protocols known collectively as TCP/IP. These include IP, the raw internet protocol (which runs on top of hardware protocols), and two protocols built on top of IP, UDP, the user datagram protocol, and TCP, the transmission control protocol.

Higher level protocols built in turn at higher levels of the network stack include DNS, the domain name system (on top of UDP), HTTP, the hypertext transfer protocol, FTP, the file transfer protocol, and SMTP, the simple mail transfer protocol (all three on top of TCP).

We have designed a simplified Maude API to DNS and TCP such that other application protocols that are built on top of TCP could be efficiently implemented in Maude. This package makes heavy use of built-in strings, which have been described in Section 3.1.

There are three basic ideas behind the implementation of built-in sockets. First, we include a system object which can be sent messages and can create other special objects and return handles (object identifiers) to them. Second, most system calls (or sequences of system calls) will be wrapped as a pair of messages, one to initiate the call (or sequence) and another to return results. For many operations this provides the sequentialization that is necessary and is inherent in system calls from a sequential language such as C. And third, multiple APIs to TCP/IP exist but AF_INET sockets are available on almost every platform. Thus we provide the socket-level interface as the basic common mechanism for internet programming in Maude.

Appendix C includes an example of how a user might use the built-in Maude socket interface to build a very simple minded HTTP/1.0 client for retrieving web pages from an arbitrary location on the internet.

We believe that Maude with built-in support for sockets will provide a powerful platform for internet programming, offering substantial advantages over the current generation of interpreted multithreaded languages such as Java [12] and Python [24]. We are currently studying the merits of a Maude-specific transfer protocol, built on top of TCP and called Mobile Object Transfer Protocol (MOTP), to be used in Mobile Maude, a mobile language extension of Maude 2.0 [9].

4 The New `META-LEVEL`

Besides allowing the metarepresentation of the extended module syntax and the new built-in modules, the two key new advantages of the new `META-LEVEL` are: a simpler representation of terms, and a richer set of *descent functions* [4].

4.1 Sort and Kind Representation

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 .
```

4.2 Term Representation

The simpler representation of terms is 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 then 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, with constants and variables being the basic cases in the construction.

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

In this way, constants and variables are metarepresented by atomic entities, namely, by special types of quoted identifiers; any further level of metarepresentation does not increase their size: it just adds one more quote. For example, the term `s(N:Nat) + 0` in a module `NAT` is now metarepresented by

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

and meta-metarepresented by

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

There are two special operations to construct (Boolean) terms, corresponding to membership test and membership lazy test:

```
op _::_ : Term Sort -> Term .
op _:::_ : Term Sort -> Term .
```

4.3 Module Representation

Modules are essentially metarepresented as explained in [4], with the following differences: we need syntax for the new conditions in conditional memberships, equations and rules; there is no need for variable declarations; and terms use now the new metarepresentation for terms.

```
op fmod_is_sorts_._____endfm : Qid ImportList SortSet SubsortDeclSet
       OpDeclSet MembAxSet EquationSet -> FModule .

op mod_is_sorts_._______endm : Qid ImportList SortSet SubsortDeclSet
     OpDeclSet MembAxSet EquationSet RuleSet -> Module .

subsort Sort < SortSet .
op (op_:_->_[_].) : Qid TypeList Type AttrSet -> OpDecl .

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

op mb_:_. : Term Sort -> MembAx [ctor] .
op cmb_:_if_. : Term Sort EqCondition -> MembAx [ctor] .
op eq_=_. : Term Term -> Equation [ctor] .
op ceq_=_if_. : Term Term EqCondition -> Equation [ctor] .
op rl[_]:_=>_. : Qid Term Term -> Rule [ctor] .
op crl[_]_=>_if_. : Qid Term Term Condition -> Rule [ctor] .
```

As a very 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 [comm] .      op ’_+_ : ’Nat ’Nat -> ’Nat [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
```

4.4 Descent Functions

We have added to the previous descent functions `metaApply`, `metaReduce`, and `metaRewrite`, a new function `metaMatch`, in order to achieve the metalevel equivalent of matching a pattern to a subject term at the top, and generalized functions `metaXapply` and `metaXmatch` that, respectively, apply a rule or attempt a match not only at the top of the subject term, but anywhere in it, and furthermore with extension.

The function `metaApply` intuitively applies a rule (whose label is the third argument) in a given module (first argument) to the top of a term (second argument) instantiated with a given substitution (fourth argument). Since there may be several possible matches due to the structural axioms of operators, the last argument is used to enumerate such matches. The result is a term, with the corresponding sort or kind, and the matching substitution.

The function `metaXapply` does the same, but using extension (see [6, Section 5.8]) and in any possible position, not only at the top. The last argument indicates the depth in the flattened term (with respect to its associative or associative-commutative operators) where the application of the rule can take place. The result has an additional component, giving the context inside the given term, where the rewriting has taken place.

```
op metaApply : Module Term Qid Substitution MachineInt
            -> [ResultTriple] .
op metaXapply : Module Term Qid Substitution MachineInt MachineInt
             -> [Result4Tuple] .
op {_,_,_} : Term Type Substitution -> ResultTriple [ctor] .
op {_,_,_,_} : Term Type Substitution Context -> Result4Tuple [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. It the matching attempt is successful, the result is the corresponding substitution. The generalization to `metaXmatch` follows exactly the same ideas as before.

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

The descent functions `metaReduce` and `metaRewrite` have not changed from the previous version, but their result includes now the sort or kind of the resulting term.

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

We are also considering adding functions `metaUnify` for unification, and `metaSearch` for built-in search.

5 Miscellaneous Features

5.1 Maude Module System

The Maude module system in Core Maude has been completely rewritten in order to support lazy flattening and lazy reparsing, and is already available in Maude 1.0.5. Lazy flattening means that when a user enters a heavily structured Maude module, the signatures are flattened eagerly as usual, but the memberships, equations, and rules are only flattened in the current module when a rewriting command is entered. This saves a lot of memory and speeds up module entry, but: (a) certain errors in memberships, equations, and rules may not be caught until the flattening takes place; (b) there is a short pause before the first execution of a rewriting command in a module takes place, while flattening is done; and (c) the order of equations and rules occurring in different modules has changed. A module's own equations and rules are now before rather than after the imported ones. Of course the user should not be relying on this.

Lazy reparsing means that when a module with the same name as an existing module is entered, it replaces the old module, and all modules that depended on the old module are marked as outdated, and will be reparsed if an attempt is made to use them. The old module's memory is freed up.

5.2 LaTeX Pretty Printing

Maude 2.0 supports direct pretty printing into LaTeX. LaTeX logging can be turned on or off by the commands `set print latex on .` and `set print latex off .`

By default, keywords are set in bold face, and terms are typeset in math mode with various conventions for generating LaTeX special symbols from their multi-character ASCII versions, for example: x2 becomes x_2, `>=` becomes $\geq$, and `=/=` becomes $\neq$. LaTeX macros may be specified for each operator with the `latex` attribute, for example:

```
op definiteIntegral : Exp Exp Exp Var -> Exp
                      [latex ({\int_{#1}^{#2} {#3} d{#4}})] .
```

Lexical analysis is done using LaTeX conventions within delimiters; the resulting expanded string is then defined by LaTeX `newcommand`. In this way, Maude 2.0 supports pretty printing into LaTeX automatically.

6 The Maude Compiler

The Maude compiler is in prototype development, but is already able to support a large subset of the language accepted by the Maude interpreter. The coverage of the compiler will increase as compilation technology is developed. The Maude compiler is implemented by compiling Maude 2.0 programs to C++, with one C++ function per Maude function, which is then compiled with GNU g++ to obtain a native executable. The current compiler can reach up to 15 million rewrites per second on a 667MHz Xeon on some examples, a factor of 5 speedup over the 2.98 million rewrites per second reached by the interpreter on the same hardware.

In the compiler, many optimizations have been made which dramatically increase performance, including:

- Sort computations and tests are optimized based on sort structure; there is almost zero cost for many-sorted modules.
- Many-to-one discrimination nets are used for outer free function symbol skeleton. We adopt a "one variable, one use" policy: no variable is reused or used in two different branches (unless it gets its value in a common parent branch) of the discrimination net code. This is important because current GNU g++ is not capable of splitting live variable ranges, which is important on Intel x86-like architectures.
- Nodes representing term graphs are of variable size (compared with fixed size dag nodes in the interpreter) because we do not need to be able to overwrite one dag node with another (in-place replacement) and we do not have to worry about fragmentation, since the generational scavenging garbage collector can move nodes to defragment memory.
- Noncanonical term representations are under consideration for various equational theories such as AC, ACU, A, and AU to avoid the linear factor overheads incurred when matching and normalizing naive flattened representation. This representation will incur a cost on term comparison, but the benefits may outweigh the costs.

The Maude compiler is integrated into the interpreter; the `creduce` (`cred`) command invokes the compiler to compile the current module (if it has not already been compiled) and passes the term to be reduced to the resulting binary executable.

7 Conclusions

We have presented the design and main new features of Maude 2.0. As already mentioned, the main design goals are: (i) greater generality and expressiveness; (ii) efficient support for a wider range of programming applications; and (iii) usability of Maude 2.0 as a key component for developing internet programming and mobile computing systems.

We view our work as a further step in the collective effort of demonstrating in practice that the rewriting logic paradigm is a very attractive candidate not just for executable specification purposes, but also for application programming, including parallel, distributed, and mobile programming. Therefore, our work should be viewed in the context of earlier work on parallel computing with rewriting languages [11,14,15].

References

[1] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An alternative to strings. *Software Practice and Experience*, 25(12):1315, 1995. `http://www.sgi.com/Technology/STL/Rope.html`

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

[3] M. Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In C. Kirchner and H. Kirchner, editors, *Proc. Second Int. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, France*, ENTCS 15. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. SRI International, January 1999. `http://maude.csl.sri.com`

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. A tutorial on Maude. SRI International, March 2000. `http://maude.csl.sri.com`

[7] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. First Int. Workshop on Rewriting Logic and its Applications, Asilomar, California*, ENTCS 4. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`

[8] G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *Proc. DARPA Information Survivability Conference and Exposition DICEX 2000, Vol. 1, Hilton Head, South Carolina*, pages 251–265. IEEE, 2000.

[9] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *Proc. Second Int. Symp. on Agent Systems and Applications/Fourth Int. Symp. on Mobile Agents, Zurich, Switzerland*, LNCS. Springer, 2000. `http://maude.csl.sri.com`

[10] F. Durán and J. Meseguer. On parameterized theories and views for Maude. In K. Futatsugi, editor, *Proc. Third Int. Workshop on Rewriting Logic and its Applications, Kanazawa, Japan*, ENTCS 36. Elsevier, 2000. This volume.

[11] J. Goguen, C. Kirchner, and J. Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proc. Workshop on Graph Reduction, Santa Fe, New Mexico*, LNCS 279, pages 53–93. Springer, 1987.

[12] E. Rusty Harold. *Java Network Programming.* O'Reilly, 1997. `http://java.sun.com`

[13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

[14] C. Kirchner and P. Viry. Implementing Parallel Rewriting. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems*, LNAI 590, pages 123–138. Springer, 1992.

[15] P. Lincoln, N. Martí-Oliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G.E. Blelloch *et al.*, *Specification of Parallel Algorithms*, pages 309–339. AMS, 1994.

[16] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Int. Workshop on Rewriting Logic and its Applications, Asilomar, California*, ENTCS 4. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`

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

[18] 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 Object-Based Concurrency*, pages 314–390. The MIT Press, 1993.

[19] J. Meseguer. Rewriting logic and Maude: A wide-spectrum semantic framework for object-based distributed systems. In *Proc. FMOODS 2000*, Kluwer, 2000.

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

[21] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *Proc. 6th Int. Conf. on Rewriting Techniques and Applications, Kaiserslautern*, LNCS 914, pages 179–193. Springer, 1995.

[22] A. Verdejo and N. Martí-Oliet. *Executing and verifying CCS in Maude.* Technical Report 99–00, Depto. de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain, February 2000.

[23] 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*, LNCS 817, pages 648–660. Springer, 1994.

[24] A. Watters, G. van Rossum, and J. C. Ahlstrom. *Internet Programming with Python.* MIS Press/Henry Holt, 1996. `http://www.python.org`

A More details of two examples

A.1 Graph Example

```
fmod A-GRAPH is
   sorts Edge Node .
   ops n1 n2 n3 n4 n5 : -> Node [ctor] .
   ops a b c d e f : -> Edge [ctor] .
   ops s t : Edge -> Node .
   eq s(a) = n1 .  eq t(a) = n2 .
   eq s(b) = n1 .  eq t(b) = n3 .
   eq s(c) = n3 .  eq t(c) = n4 .
   eq s(d) = n4 .  eq t(d) = n2 .
   eq s(e) = n2 .  eq t(e) = n5 .
   eq s(f) = n2 .  eq t(f) = n1 .
endfm
```

A.2 CCS Syntax

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

fmod PROCESS is protecting ACTION .
   sorts ProcessId Process .
   subsorts Qid < ProcessId < Process .
   op 0 : -> Process .                              *** inaction
   op _._ : Act Process -> Process [prec 25] .      *** prefix
   op _+_ : Process Process -> Process [assoc comm prec 35] .
            *** summation
   op _|_ : Process Process -> Process [assoc comm prec 30] .
            *** composition
   op _\_ : Process Label -> Process [prec 20] .    *** restriction
   op _[_/_] : Process Label Label -> Process [prec 20] .
               *** relabelling: [b/a] relabels "a" to "b"
endfm

fmod CCS-CONTEXT is protecting PROCESS .
   sorts BadProcess Context BadContext .
   subsort Process < BadProcess .
   subsort Context < BadContext .
   op _=def_ : ProcessId Process -> Context [prec 40] .
   op nil : -> Context .
   op _&_ : BadContext BadContext -> BadContext
            [assoc comm id: nil prec 42] .
   op _definedIn_ : ProcessId Context -> Bool .
   op def : ProcessId Context -> BadProcess .
   op not-defined : -> BadProcess .
   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' = true .
endfm
```

B Built-in String Functions

The following table briefly describes the built-in functionality provided for strings, in particular standard strings of 8-bit characters. The current implementation of built-in strings includes the following functions:

`ascii(C)`	returns the ASCII code of `C`
`char(M)`	returns the character with code `M` if `M` in `0..255`
`length(S)`	is the length of `S`
`S + T`	forms the concatenation of `S` and `T`
`substr(S, P, L)`	is the substring of `S`, starting at `P` of length `L`
`find(S, T, P)`	returns the position of the first occurrence of `T` in `S` starting from position `P` or `notFound`
`rfind(S, T, P)`	returns the position of the first occurrence of `T` in `S` searching backwards from position `P` or `notFound`
`S < T`	Lexicographic ordering on strings
`S > T`	
`S <= T`	
`S => T`	
`string(M, B)`	returns the string representation of `M` in base `B` if `B` is in the range `2..36` (`a..z` are used as digits)
`machineInt(S, B)`	returns the machine integer represented by `S` in base `B` if in fact `S` does represent a machine integer in base `B` where `B` is in the range `2..36` (`a..z` and `A..Z` are recognized as digits)

C Internet Example: HTTP Client

The following example uses built-in string functions and built-in socket interfaces to build a very simple minded HTTP/1.0 client for retrieving web pages from an arbitrary location on the internet.

```
omod HTTP/1.0-CLIENT is
***
*** Simple HTTP/1.0 client. Urls must be of form "hostname" or
***     "hostname/path" e.g. maude.csl.sri.com/new.html
*** Virtual hosts supported. No error handling.
***
   inc TCP .
   sort State .
   ops idle dnsLookup connecting sending receiving : -> State [ctor] .
   class HttpClient | state : State, requester : Oid, url : String .
   sort HttpClientId .
   subsort HttpClientId < Oid .
   msg getPage : HttpClientId Oid String -> Msg .
   msg gotPage : Oid HttpClientId String String -> Msg .

   var H : HttpClientId .   var R : Oid .
   vars C S Url : String .  var Aliases : StringList .
   var Ad : Address .       var Rest : AddressList .

   op extractHostName : String -> String .
   op extractPath : String -> String .
   op extractHeader : String -> String .
   op extractBody : String -> String .

   eq extractHostName(S) = if find(S, "/", 0) == notFound then S
                           else substr(S, 0, find(S, "/", 0)) fi .

   eq extractPath(S) = if find(S, "/", 0) == notFound then "/"
                       else substr(S, find(S, "/", 0), length(S)) fi .

   eq extractHeader(S) = substr(S, 0, find(S, "\r\n\r\n", 0) + 2) .
   eq extractBody(S) = substr(S, find(S, "\r\n\r\n", 0), length(S)) .

   rl getPage(H, R, Url)
      < H : HttpClient | state : idle >
   => < H : HttpClient | state : dnsLookup, requester : R, url : Url >
      getHostByName(System, H, extractHostName(Url)) .

   rl gotHostEntry(H, System, [C, Aliases, Ad Rest])
      < H : HttpClient | state : dnsLookup >
   => < H : HttpClient | state : connecting >
      createTcpClientSocket(System, H, Ad, 80) .

   rl createdTcpClientSocket(H, System, TS)
      < H : HttpClient | state : connecting, url : Url >
   => < H : HttpClient | state : sending, url : Url >
      send(TS, H, "GET " + extractPath(Url) + " HTTP/1.0\r\nHost: " +
    extractHostName(Url) + "\r\n\r\n") .

   rl sent(H, TS)
      < H : HttpClient | state : sending >
   => < H : HttpClient | state : receiving >
      receive(TS, H, MAX-MACHINE-INT, MAX-MACHINE-INT) .

   rl received(H, TS, S)
      < H : HttpClient | state : receiving, requester : R, url : Url >
   => < H : HttpClient | state : idle >
      gotPage(R, H, extractHeader(S), extractBody(S))
      closeSocket(TS) .
endom
```