The Maude 2.0 System*

Manuel Clavel[1], Francisco Durán[2], Steven Eker[3], Patrick Lincoln[3],
Narciso Martí-Oliet[1], José Meseguer[4], and Carolyn Talcott[3]

[1] Universidad Complutense de Madrid, Spain
[2] Universidad de Málaga, Spain
[3] SRI International, CA, USA
[4] University of Illinois at Urbana-Champaign, IL, USA

Abstract. This paper gives an overview of the Maude 2.0 system. We emphasize the full generality with which rewriting logic and membership equational logic are supported, operational semantics issues, the new built-in modules, the more general Full Maude module algebra, the new `META-LEVEL` module, the LTL model checker, and new implementation techniques yielding substantial performance improvements in rewriting *modulo*. We also comment on Maude's formal tool environment and on applications.

1 Introduction

Rewriting logic has been shown to have good properties as a semantic and logical framework [20]. The computational and logical meanings of a rewrite $t \longrightarrow t'$ are like two sides of the same coin. Computationally, $t \longrightarrow t'$ means that the state component t can *evolve* to the component t'. Logically, $t \longrightarrow t'$ means that from the formula t one can *deduce* the formula t'. Furthermore, rewriting logic has been shown to have good properties not only for specification, but also as a declarative programming paradigm, as demonstrated by the mature implementations of the ELAN [1], CafeOBJ [15], and Maude [6] languages.

We will focus in this paper on the main new features in Maude 2.0. We refer the reader to [4,5] for details on previous releases. Given space limitations, not even all these new features can be discussed here. The Maude system, its documentation, and related papers and applications are available from the Maude website `http://maude.cs.uiuc.edu`.

The Maude 2.0 system supports both equational and rewriting logic computation with high generality and expressiveness, yet without compromising performance. Functional modules are membership equational theories, whereas system modules are very general rewrite theories whose rules can have equations, memberships, and rewrites in their conditions, and where some operator arguments can be *frozen* to block undesired rewrites (see Section 2). Furthermore, Full Maude 2.0 supports parameterized modules, theories, and views, and object-oriented modules. Besides supporting equational simplification, Maude 2.0 supports several fair rewriting strategies as well as breadth-first search. Reflective capabilities are substantially extended in a new `META-LEVEL` module. There are also efficient predefined implementations of useful arithmetic and

* Research supported by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130, ONR Grant N00014-02-1-0715, NSF grants CCR-9900326, CCR-0234603 and CCR-0234524, and by CICYT projects TIC 2000–0701–C02–01 and TIC 2001–2705–C03–02.

string data types. Since rewrite theories are ideally suited for specifying concurrent systems, Maude 2.0 supports efficient explicit-state model checking of linear temporal logic (LTL) properties satisfied by finite-state rewrite theories. The efficiency of rewriting modulo axioms has also been increased thanks to some novel implementation techniques. Finally, using reflection an environment of formal tools for Maude 2.0, extending earlier tools, is currently under development.

The structure of this document is as follows. Section 2 discusses the semantics of Maude 2.0. Section 3 presents some of the new features in this release. Section 4 is dedicated to the implementation and performance of the system. Section 5 comments on Maude's formal tool environment. Finally, Section 6 draws some concluding remarks.

2 Generalized Logical and Operational Semantics

The close contact with many specification and programming applications has served as a good stimulus for a substantial increase in expressive power of the rewriting logic formalism in general, and of its Maude realization in particular. Specifically, Maude 2.0 supports rewriting logic computation generalized along three key dimensions. A first dimension concerns the generality of the underlying equational logic. Since a rewrite theory is essentially a triple $\mathcal{R} = (\Sigma, E, R)$, with (Σ, E) an equational theory, and R a set of labeled rewrite rules that are applied *modulo* the equations E, the more general the underlying equational logic, the more expressive the rewriting logic. Maude 2.0's underlying equational logic is *membership equational logic* [22], a very expressive many-kinded Horn logic whose atomic formulas are equations $t = t'$ and memberships $t : s$, stating that a term t has sort s. A second dimension concerns the generality of conditions in conditional rewrite rules that can be of the form,

$$(\forall X)\ r{:}t \rightarrow t' \text{ if } \bigwedge_{i \in I} p_i = q_i \ \wedge\ \bigwedge_{j \in J} w_j : s_j \ \wedge\ \bigwedge_{l \in L} t_l \rightarrow t'_l$$

where r is the rule label, all terms are Σ-terms, and the rule can be made conditional to other equations, memberships, and rewrites being satisfied. A third dimension involves support for declaring certain operator arguments as *frozen*, thus blocking rewriting under them. Therefore, a Maude (system) module is a *generalized rewrite theory*, defined as a 4-tuple $\mathcal{R} = (\Sigma, E, \phi, R)$, where (Σ, E) is a membership equational theory, R is a set of labeled conditional rewrite rules of the general form above, and ϕ is a function assigning to each operator $f : k_1 \ldots k_n \rightarrow k$ in Σ the subset $\phi(f) \subseteq \{1, \ldots, n\}$ of its frozen arguments. In Maude, membership equational theories define the equational sublanguage of *functional modules*.

Unfrozen arguments (those not frozen) are for rewrite theories the analog of the arguments specified in *evaluation strategies* [10] used for equational theories in OBJ, CafeOBJ, and Maude to improve efficiency and/or to guarantee the termination of computations, replacing unrestricted equational rewriting by context-sensitive rewriting [19]. Thus, in Maude 2.0 rewriting with both equations E and rules R can be made context-sensitive. The mathematical semantics of generalized rewrite theories, and thus of modules in Maude 2.0, has been recently developed by Bruni and Meseguer [2], who have given generalized rules of deduction, and have shown the existence of initial and free models and the completeness of rewriting logic deduction relative to the generalized model theory.

There is yet another way in which Maude 2.0 supports rewriting logic and its underlying membership equational logic in its fullest possible generality, namely by the way executability issues are dealt with in the language. The point, of course, is that efficient and complete computation by rewriting is not possible for arbitrary equational theories, unless they satisfy good properties such as confluence, sort-decreasingness, and perhaps termination. Similarly, to be efficiently executable, a generalized rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ should first of all have (Σ, E) satisfying the above executability requirements, and should furthermore be *coherent* [27]. Executability is of course what we want for programming; but it is too restrictive for specification, transformation, and reasoning purposes, even when programming is the ultimate goal. For this reason, in Full Maude (as in OBJ) there is a linguistic distinction between *modules*, that are typically used for programming as executable theories, and *theories*, which need not be executable and are used for specification purposes (for example, to specify the semantic requirements of interfaces in *parameterized* modules). Maude 2.0 supports specification of arbitrary membership equational logic theories and of arbitrary rewrite theories, while at the same time keeping a sharp distinction between executable and non-executable statements (i.e., equations, memberships, or rules). This distinction is achieved by means of the `nonexec` attribute, with which such statements can be labeled. In fact, in Maude 2.0 both modules and theories can be either: (1) fully executable, or (2) partially executable (some statements are `nonexec`), or (3) non-executable (all statements are `nonexec`). Fully executable equational and rewrite theories are called *admissible*, and satisfy the above-mentioned executability requirements; however, in keeping with the desired generality of conditions in equations and rules, extra variables can appear in conditions, provided that they are only introduced by patterns in *matching equations* or in the righthand sides of rewrites (see Section 3.2). Nevertheless, executability is a relative matter. In Maude 2.0 all statements are executable at the metalevel, using reflection and the `META-LEVEL` module (see Section 3.5) but non-executable ones will need strategies to guide their metalevel execution. This support for a disciplined coexistence of executable and non-executable statements allows not only a seamless integration of specification and code, but also a seamless integration of Maude with its formal tools (see Section 5).

3 Some New Features in Maude 2.0

Maude 2.0 presents a number of new features with respect to previous releases. In the following sections we shall discuss some of the most relevant ones, namely, the possibility of accessing the kinds, the new form of conditions in conditional statements, a search facility for doing breadth first search with cycle detection, the new built-in modules, the new possibilities for parameterized programming, the new metalevel, and the LTL model checker. Other features not discussed here include: a rule and position fair strategy, on-the-fly declaration of variables, statement attributes (all statements can be given labels for improving tracing, and we can attach an arbitrary string of metadata to a statement for metaprocessing), the possibility of using efficiently huge towers of unary operator symbols, facilities for improving pretty-printing of terms and for identifying possible incompleteness of specifications, new profiling and debugging features, and so on.

3.1 Access to Kinds

A membership equational signature Σ has a set K of *kinds*, and for each $k \in K$ a set S_k of *sorts* of that kind. Maude does automatic kind inference from the sorts declared by the user and their subsort relations, but kinds are not explicitly named; instead, a kind k is identified with the set S_k of its sorts, interpreted as an *equivalence class* modulo the equivalence relation generated by the subsort ordering. 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.

Let us assume a graph specification with sorts `Node` and `Edge` and operations `source` and `target` giving, respectively, the source and target nodes of each edge, as well as specific edge and node constants. Then, we extend such a specification by declaring a sort `Path` of paths over the graph, together with a *partial* concatenation operator, and appropriate source and target functions over paths as follows, where the subsort declaration states that edges are paths of length one.

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

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

Given variables `E` and `P` of sorts `Edge` and `Path`, respectively, we may express the condition defining path concatenation with the conditional membership axiom

```
cmb E ; P : Path if target(E) = source(P) .
```

stating that an edge concatenated with a path is also a path when the target node of the edge coincides with the source node of the path. This has the effect of defining path concatenation as a partial function on paths, although it is total on the kind `[Path]` of “confused paths.”

3.2 More Expressive Conditions in Conditional Statements and Searching

Equational conditions in conditional equations and memberships are made up from individual equations $t = t'$ and memberships $t : s$ using a binary conjunction connective `/\` which is assumed associative. Furthermore, equations in conditions have two variants, namely, ordinary equations `t = t'`, and *matching equations* `t := t'`. For example, assuming a variable `E` of sort `Edge`, and variables `P` and `S` of sort `Path`, the source function over paths may be defined by means of matching equations in conditions as follows:

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

Matching equations are mathematically interpreted as ordinary equations; however, operationally they are treated in a special way and they must satisfy special requirements. Note that the variables `E` and `S` in the above matching equation do not appear in the lefthand sides of the corresponding conditional equation. In the execution of this equation, 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* [6].

The satisfaction of the conditions is attempted sequentially from left to right. Since matching takes place *modulo* equational attributes, in general many different matches may have to be tried until a match of all the variables satisfying the condition is found. All conditional equations in a functional module have to satisfy certain *admissibility requirements*, ensuring that all the extra variables will become instantiated by matching (see [6] for details).

Conditional rewrite rules can take the most general possible form in the variant of rewriting logic built on top of membership equational logic, as explained in Section 2, 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. Of course, in that full generality the execution of a rewrite theory specified as 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]. However, a quite general class of system modules, called *admissible modules*, are executable by Maude's interpreter using its built-in strategies. A system module M is called *admissible* if its underlying equational theory is confluent, sort decreasing and terminating, its rules are coherent with respect to its equations, and each of its rewrite rules satisfies certain requirements ensuring that all the extra variables will become instantiated [6].

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

Searching is also available to the user through the `search` command, which looks for all the rewrites of a given term that match a given pattern satisfying some condition. When a search command terminates, either because there was a finite state graph, or because a limit to the number of solutions was given, the state graph is retained in memory. It is then possible to obtain the whole generated search graph and to interrogate the state graph for the path from the start term to any reachable state.

3.3 Built-in Funtional Modules

Maude 2.0 includes some built-in functional modules providing convenient high-performance functionality within the Maude system. In particular, the built-in modules of integers, natural, rational and floating-point numbers, quoted identifiers, and strings provide a minimal set of efficient operations for Maude programmers.

The built-in natural numbers allow Maude programmers to deal with natural numbers with a C-like performance for simple arithmetic operations on them (using GNU GMP). 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 allows programmers to manipulate numbers as if they were represented with explicit successor notation, and to reflect those numbers up to the metalevel. Integers are constructed from natural numbers using the unary minus operator. Similarly, the rational numbers are constructed from natural numbers using a division operator. 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. Floats are not algebraic term structures; they are treated as a large set of constants.

Maude's built-in strings are based on the SGI rope package which has been optimized for functional programming, where copying with modification is supported efficiently, while arbitrary in-place updates are not. The Maude string package is compatible with the `QID` built-in module [6] of quoted identifiers, and interoperates with the Maude 2.0 scheme for metarepresenting user constants. A number of conversion functions is also provided.

3.4 Parameterized Modules and Theories

Full Maude is an extension of Maude written in Maude itself that supports an algebra of parameterized modules, views, and module expressions in the Clear/OBJ style as well as object-oriented modules with convenient syntax for object-oriented applications. We distinguish three key entities: *modules*, which are theories with an initial or 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. In Maude 2.0, by means of the Full Maude 2.0 module algebra, modules, theories, and views can all be parameterized.

By using parameterized theories and views we can instantiate parameterized theories and modules in an incremental way, gaining in flexibility. The use of parameterized views allow us, for example, to define a view `Set(X :: TRIV)` from the trivial theory `TRIV` with only one sort `Elt` to the parameterized module `SET(X :: TRIV)` mapping `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` and the module `LIST(X :: TRIV)` of lists, we can have the module `LIST(Set(Nat))` of lists of sets of natural numbers, or, given a module `STACK(X :: TRIV)` of stacks and a view `Bool` from `TRIV` to the built-in module `BOOL`, stacks of sets of booleans with `STACK(Set(Bool))`.

3.5 Reflection and the **META-LEVEL** Module

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully represented in itself.

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective [8]. In Maude, key functionality of a metalevel theory with several metalevel functions has been efficiently implemented in its functional module `META-LEVEL`. Maude 2.0 includes improvements in the metarepresentations of terms and modules, and in some of the functions already available in Maude 1.0. Moreover, Maude 2.0 also provides some new metalevel functions. Among others, `META-LEVEL` includes the following functions: (1) the process of reducing a term to normal form is reified by a function `metaReduce`; (2) the process of applying a rule to a subject term is reified by functions `metaApply` and `metaXapply`; (3) the process of rewriting a term is reified by functions `metaRewrite` and `metaFrewrite`, which use, respectively, the top-down rule fair and position fair default strategies; (4) the process of matching a pattern to a subject term is reified by functions `metaMatch` and `metaXmatch`; (5) a function `metaSearch` reifies the process of searching for a particular pattern term; and (6) parsing and pretty printing of a term, as well as key sort operations, are also reified by corresponding metalevel functions. There are also new ascent functions `upMbs`, `upEqs`, and `upRls` for obtaining the metarepresentation of membership axioms, equations, and rules of a given module in the module database.

3.6 The Maude LTL Model Checker

A model checker typically supports two different levels of specification: (1) a system specification level, in which the concurrent system to be analyzed is formalized; and (2) a property specification level, in which the properties to be model checked—for example, temporal logic formulas—are specified. The Maude LTL model checker has been designed with the goal of combining the very expressive and general system specification capabilities of Maude with an LTL model checking engine that benefits from some of the most recent advances in on-the-fly explicit-state model checking techniques.

A Maude module specifies a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$. Fixing a distinguished sort `State`, the initial model $\mathcal{T}_{\mathcal{R}}$ of $\mathcal{R}$ has an underlying Kripke structure $\mathcal{K}(\mathcal{R}, \texttt{State})$ given by the total binary relation extending its one-step sequential rewrites. To the initial algebra of states $T_{\Sigma/E}$ we can likewise associate equationally-defined computable state predicates as atomic predicates for such a Kripke structure. In this way we obtain a language of LTL properties of the rewrite theory $\mathcal{R}$.

Maude 2.0 supports on-the-fly LTL model checking for initial states $[t]$ of an admissible rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ such that the set $\{[u] \in T_{\Sigma/E} \mid \mathcal{R} \vdash [t] \rightarrow [u]\}$ of all states reachable from $[t]$ is finite. The syntax of the state predicates we wish to use is defined by means of constants and operators of sort `Prop`, a subsort of `Formula` (i.e., LTL formulas), and their semantics is defined by means of equations involving the operator `_|=_ : State Formula -> Result`. These sorts and operator are declared in the `MODEL-CHECKER` module (and its submodules). Given an initial state, of sort `State`, we can model check any LTL formula involving such predicates with two possible results: `true` if the property holds, or a counterexample expressed as a finite path followed by a cycle if it does not.

Maude offers also a LTL satisfiability decision procedure in its predefined functional module `SAT-SOLVER`, which can also be used as a tautology checker.

4 Implementation and Performance

Maude 2.0, like Maude 1.0, is implemented as a hierarchy of C++ class libraries. There are a large number of incremental improvements, but we highlight the major ones.

The Core Rewrite Engine. The most radical change from Maude 1.0 is the use of a novel term representation based on persistent data structures [9] for E-rewriting [11]. In some cases, new rewriting algorithms based on this representation can dramatically improve the rewriting speed for large terms. Table 1 compares the performance of Maude with and without this representation for the example in Appendix A on a 2.8GHz, 2GByte Intel Xeon. The example consists in a specification `MAP` for a map data type together with a test program `MAP-TEST` that uses it to compute the Fibonacci function modulo 100.

Another improvement in the core rewrite engine is left-to-right sharing, in which subterms that occur in a lefthand side pattern and are repeated in the righthand side can be shared, so that when an instance of the righthand side is constructed, the subterm matched by the lefthand side subterm can be reused. Other improvements include a new discrimination net algorithm for the free theory that takes sort information into account, and the removal of a number of bottlenecks in the full AC/ACU matching algorithm.

The Metalevel. Apart from the new metaterm representation and new metalevel functions, the big change from Maude 1.0 is improved caching to cut the cost of changing levels in common cases. As well as caching metamodules on a least recently used basis, calls to the metalevel functions that take a numeric argument specifying a solution number—such as `metaApply`, `metaXapply`, `metaSearch`, etc.—are also cached, along with the rewriting state. So a subsequent call to find the next solution can compute it incrementally starting from the old state.

		Without		With	
Problem size	Rewrites	Seconds	Rews./second	Seconds	Rews./second
1000	5999	0.25	23996	0.02	299950
10000	59999	55.72	1076	0.21	285709
100000	599999	5676.44	105	2.82	212765

Table 1. Performance in both seconds and rewrites/second with and without the persistent representation for ACU terms.

The Model Checker. On-the-fly LTL model checking is performed by constructing a Büchi automaton from the negation of the property formula and lazily searching the synchronous product of the Büchi automaton and the system state transition diagram for a reachable accepting cycle. The negated LTL formula is converted to negative normal form and heuristically simplified by a set of Maude equations, mostly derived from the simplification rules in [14,23]. Rather than the classical tableaux construction [17], we use a newer technique proposed in [16] based on very weak alternating automata to which we add some strongly connected component optimizations adapted from those in [23]. Throughout the computation, the pure propositional subformulas labeling the arcs of the various automata are stored as BDDs to allow computation of conjunctions, elimination of contradictions, and combination of parallel arcs by disjunction. We use the double-depth first method of [18] to lazily generate and search the synchronous product.

5 Formal Tools

In addition to the formal methods directly supported by Maude, one can use Maude as a formal metatool to build other formal tools supporting proofs of correctness of highly critical properties. Reflection and the flexible uses of rewriting logic as a logical framework are the key features making it easy to develop such formal tools and their user interfaces. The paper [7] gives a detailed account of a wide range of formal tools that have been defined in Maude by different authors. Among others, we may mention: the inductive theorem prover ITP tool, the coherence checker and the coherence completion tools, the Church-Rosser Checker tool, the termination checker and the Knuth-Bendix completion tools, the Real-Time Maude tool, etc. There are extensions of these tools currently under development, whose implementations will greatly benefit from the new features of Maude 2.0.

6 Concluding Remarks

The advances in Maude 2.0 have been used to good advantage in several recent applications. The Pathway Logic project uses Maude to develop and analyze biological networks [12,13]. Search and model-checking are used to explore possible execution paths, and the new descent functions are used to analyze and visualize model-checking results. The ascent functions are used to transform the Maude model into a Petri net model for further analysis of possible execution paths. Work on CCS [26] and the Pi-Calculus [25] provides additional examples of the usefulness of new features of Maude 2.0, especially frozen arguments, enriched rule conditions, search, and `metaSearch`. Search, model-checking, and rewrites in rule conditions have been used in a project to model and analyze a proposed secure architecture for accessing remote services in Java [24]. Work is in progress by two of the authors and M. Palomino using the metalevel features and formal tools to define and implement abstractions that convert infinite state models into finite state abstractions [21].

Rewriting logic and its realization in Maude allow for very natural modeling of distributed systems. The next major development of Maude will be to provide an extension that supports executable models that interact with their environment. This will build on the support for concurrent objects and object rewriting in core Maude 2.0 and allow communication with external objects using asynchronous message passing. This will provide access to internet sockets, file systems, window systems, and so on.

References

1. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.
2. R. Bruni and J. Meseguer. Generalized rewrite theories. To appear in *Procs. of ICALP'03*. LNCS. Springer, 2003.
3. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude Manual, 1999. `http://maude.cs.uiuc.edu`.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. The Maude system. In *Procs. of RTA'99*. LNCS 1631, pp. 240–243. Springer, 1999.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
7. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In *Procs. of FM'99*. LNCS 1709, pp. 1684–1703. Springer, 1999.
8. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, 2002.
9. J. R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Science*, 38:86–124, 1989.
10. S. Eker. Term rewriting with operator evaluation strategy. In *Procs. of WRLA'98*. ENTCS 15. Elsevier, 1998.
11. S. Eker. Associative-commutative rewriting on large terms. This volume.
12. S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway logic: Symbolic analysis of biological signaling. In *Procs. of the Pacific Symposium on Biocomputing*, pp. 400–412, 2002.
13. S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott. Pathway logic: Executable models of biological networks. In *Procs. of WRLA'02*. ENTCS 71. Elsevier, 2002.
14. K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Procs. of CONCUR'00*. LNCS 1877, pp. 153–167. Springer, 2000.
15. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
16. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Procs. of CAV'01*. LNCS 2102, pp. 53–65. Springer, 2001.
17. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pp. 3–18. Chapman and Hall, 1995.
18. G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *Design: An International Journal*, 13(3):289–307, 1998.
19. S. Lucas. Termination of rewriting with strategy annotations. In *Procs. of LPAR'01*. LNAI 2250, pp. 669–684. Springer, 2001.
20. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

21. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. To appear in *Procs. of CADE'03*. LNCS. Springer, 2003.
22. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Procs. of WADT'97*. LNCS 1376, pp. 18–61. Springer, 1998.
23. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Procs. of CAV'00*. LNCS 1633, pp. 247–263. Springer, 2000.
24. C. Talcott. To be presented at the DARPA FTN Winter 2003 PI meeting, TX, USA, 2003.
25. P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. In *Procs. of WRLA'02*. ENTCS 71. Elsevier, 2002.
26. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In *Procs. of WRLA'02*. ENTCS 71. Elsevier, 2002.
27. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.

A Map Benchmark Example

```
fmod MAP is
  sorts Domain Range Pair Map .
  subsort Pair < Map .
  op _|->_ : Domain Range -> Pair .
  op empty : -> Map .
  op _,_ : Map Map -> Map [assoc comm id: empty] .
  op undefined : -> [Range] .
 var D : Domain . vars R R' : Range . var M : Map .
  op _[_] : Map Domain -> [Range] .
  eq (M, D |-> R)[D] = R .
  eq M[D] = undefined [owise] .
  op insert : Domain Range Map -> Map .
  eq insert(D, R, (M, D |-> R')) = (M, D |-> R) .
  eq insert(D, R, M) = (M, D |-> R) [owise] .
endfm

fmod MAP-TEST is pr MAP . pr NAT .
  subsort Nat < Domain Range .
 var N : Nat .
  op f : Nat -> Map .
  eq f(0) = insert(0, 1, empty) .
  eq f(1) = insert(1, 1, f(0)) .
  eq f(s s N)
    = insert(s s N, ((f(s N)[s N]) + (f(s N)[N])) rem 100, f(s N)) .
endfm
```