The Maude System*

M. Clavel[1], F. Durán[2], S. Eker[2], P. Lincoln[2], N. Martí-Oliet[3], J. Meseguer[2], and J. F. Quesada[4]

[1] Department of Philosophy, University of Navarre, Spain
[2] SRI International, Menlo Park, CA 94025, USA
[3] Facultad de Ciencias Matemáticas, Universidad Complutense, Madrid, Spain
[4] CICA (Centro de Informática Científica de Andalucía), Seville, Spain

1 Introduction

Maude is a high-performance language and system supporting both equational and rewriting logic computation for a wide range of applications, including development of theorem proving tools, language prototyping, executable specification and analysis of concurrent and distributed systems, and logical framework applications in which other logics are represented, translated, and executed.

Maude's *functional modules* are theories in *membership equational logic* [8,1], a Horn logic whose atomic sentences are either equalities $t = t'$ or *membership assertions* of the form $t : s$, stating that a term t has a certain sort s. Such a logic extends OBJ3's [4] order-sorted equational logic and supports sorts, subsorts, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains. Maude's functional modules are assumed to be Church-Rosser; they are executed by the Maude engine according to the rewriting techniques and operational semantics developed in [1].

Membership equational logic is a sublogic of *rewriting logic* [6]. A rewrite theory is a pair (T, R) with T a membership equational theory, and R a collection of labeled and possibly conditional *rewrite rules* involving terms in the signature of T. Maude's *system modules* are rewrite theories in exactly this sense. The rewrite rules $r : t \longrightarrow t'$ in R are *not* equations. Computationally, they are interpreted as local *transition rules* in a possibly concurrent system. Logically, they are interpreted as *inference rules* in a logical system. This makes rewriting logic both a general *semantic framework* to specify concurrent systems and languages [7], and a general *logical framework* to represent and execute different logics [5].

Rewriting in (T, R) happens *modulo* the equational axioms in T. Maude supports rewriting modulo different combinations of associativity, commutativity, identity, and idempotency axioms. The rules in R need not be Church-Rosser and need not be terminating. Many different rewriting paths are then possible; therefore, the choice of appropriate *strategies* is crucial for executing rewrite theories. In Maude, such strategies are not an extra-logical part of the language.

* Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grant CCR-9633363.

They are instead *internal strategies* defined by rewrite theories at the metalevel. This is because rewriting logic is *reflective* [2] in the precise sense of having a *universal theory* U that can represent any finitely presented rewrite theory T (including U itself) and any terms t, t' in T as terms $\overline{T}$ and $\overline{t}, \overline{t'}$ in U, so that we have the following equivalence

$$T \vdash t \longrightarrow t' \;\;\Leftrightarrow\;\; U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle.$$

Since U is representable in itself, we can then achieve a "reflective tower" with an arbitrary number of levels of reflection. Maude efficiently supports this reflective tower through its `META-LEVEL` module, which makes possible not only the declarative definition and execution of user-definable rewriting strategies, but also many other applications, including an extensible *module algebra* of parameterized module operations that is defined and executed within the logic.

This extensibility by reflection is exploited in Maude's design and implementation. Core Maude (Section 2) supports module hierarchies consisting of (unparameterized) functional and system modules and provides the `META-LEVEL` module. Full Maude (Section 3) is an extension of Core Maude written in Core Maude itself that supports a module algebra of parameterized modules, views, and module expressions in the OBJ style [4] as well as *object-oriented modules* with convenient syntax for object-oriented applications. The paper ends with a summary of different applications (Section 4). The Maude 1.0 system and its documentation have been available for distribution (free of charge) since January 1999 through the Maude web page `http://maude.csl.sri.com`.

2 Core Maude

The Maude system is built around the Core Maude interpreter, which accepts module hierarchies of (unparameterized) functional and system modules with user-definable mixfix syntax. It is implemented in C++ and consists of two parts: the rewrite engine, and the mixfix frontend.

The rewrite engine is highly modular and does not contain any Maude-specific code. Two key components are the "core" module and the "interface" module. The core module contains classes for objects which are not specific to an equational theory, such as equations, rules, sorts, and connected sort components. The "interface" module contains abstract base classes for objects that may have a different representation in different equational theories, such as symbols, term nodes, dag nodes, and matching automata. New equational theories can be "plugged in" by deriving from the classes in the "interface" module. To date, all combinations of associativity, commutativity, left and right identity and idempotence have been implemented apart from those that contain both associativity and idempotence. New built in symbols with special rewriting (equation or rule) semantics may be easily added.

The engine is designed to provide the look and feel of an interpreter with hooks for source level tracing/debugging and user interrupt handling. These goals prevent a number of optimizations that one would normally implement in a compiler, such as transforming the user's term rewriting system, or keeping pending evaluations on a stack and only building reduced terms. The actual implementation is a semi-compiler where the term rewriting system is compiled to a system of tables and automata, which is then interpreted. Typical performance with the current version is 800K-840K free-theory rewrites per second and 27K-111K associative-commutative (AC) rewrites per second on standard hardware (300MHz Pentium II). The figure for AC rewriting is highly dependent on the complexity of the AC patterns (AC matching is NP-complete) and the size of the AC subjects. The above results were obtained using fairly simple linear and non-linear patterns and large (hundreds of nested AC operators) subjects.

The mixfix frontend consists of a bison/flex-based parser for Maude's surface syntax, a grammar generator (which generates the context free grammar for the user-definable mixfix syntax in a module together with some built-in extensions), a context free parser generator, a mixfix pretty printer, a fully reentrant debugger, the built-in functions for quoted identifiers and the `META-LEVEL` module, together with a considerable amount of "glue" code holding everything together. Many of the C++ classes are derived from those in the rewrite engine. The Maude parser generator (MSCP) is implemented using SCP as the formal kernel [9]. The techniques used include β-extended GFGs (that is, CFGs extended with "bubbles" (strings of tokens) and precedence/gather patterns). MSCP provides efficient treatment of syntactic reflection, and a basis for flexible syntax definition.

In Maude, key functionality of the universal theory U has been efficiently implemented in the functional module `META-LEVEL`. In `META-LEVEL` Maude terms are reified as elements of a data type `Term`, and Maude modules are reified as terms in a data type `Module`. The processes of reducing a term to normal form in a functional module and of rewriting a term in a system module using Maude's default interpreter are respectively reified by functions `meta-reduce` and `meta-rewrite`. Similarly, the process of applying a rule of a system module to a subject term is reified by a function `meta-apply`. Furthermore, parsing and pretty printing of a term in a signature, as well as key sort operations are also reified by corresponding metalevel functions.

3 Full Maude

Using reflection Core Maude can be extended to a much richer language with an extensible *module algebra* of module operations that can make Maude modules highly reusable. The basic idea is that the `META-LEVEL` module can be extended with new data types—extending the `Module` sort of flat modules to structured and parameterized modules—and with new functions corresponding to new module operations—such as instantiation of parameterized modules by views, flattening of module hierarchies into single modules, desugaring of object-oriented modules into system modules, and so on. All such new types and operations can be defined in Core Maude. Using the meta-parsing and meta-pretty printing functions in `META-LEVEL` and a simple `LOOP-MODE` module providing input/output we have developed in Core Maude a user interface for Full Maude. At present, Full Maude supports all of Core Maude plus *object-oriented* modules, *parameterized* modules, *theories* with loose semantics to state formal requirements in parameters, *views* to bind parameter theories to their instances, and *module expressions* instantiating and composing parameterized modules.

4 Applications

Maude is an attractive formal meta-tool for building many advanced applications and formal tools. The largest application so far is Full Maude (7,000 lines of Maude code). Other substantial applications include: an inductive theorem prover; a Church-Rosser checker (both part of a formal environment for Maude and for the CafeOBJ language [3]); an HOL to Nuprl translator; and a translator from J. Millen's CAPSL specification language to the CIL intermediate language. In addition, several language interpreters and strategy languages, several object-oriented specifications—including cryptographic protocols and network applications—and a variety of executable translations mapping logics, architectural description languages and models of computation into the rewriting logic reflective framework have been developed by different authors.

References

1. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. To appear in *Theoretical Computer Science*.
2. M. Clavel. Reflection in general logics and in rewriting logic, with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.
3. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational logic tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.
4. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen and G.R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1998.
5. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
6. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
7. J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. CONCUR'96, Pisa, August 1996*, pages 331–372. Springer LNCS 1119, 1996.
8. J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., Proc. WADT'97, 18–61, Springer LNCS 1376, 1998.
9. J. Quesada. *The SCP parsing algorithm based on syntactic constraint propagation.* Ph.D. thesis, University of Seville, 1997.